3 Features, Scenarios, and Stories

Some software products are inspired. The developers of these products have a vision of the software that they want to create. They don’t have a product manager, they don’t do user surveys, they don’t collect and document requirements or model how users will interact with the system. They simply get on with developing a prototype system. Some of the most successful software products, such as Facebook, started like this.

However, the vast majority of software products that are solely based on a developer’s inspiration are commercial failures. These products either don’t meet a real user need or don’t fit with the ways in which users really work. Inspiration is important but most successful products are based on an understanding of business and user problems and user interaction. Even when inspiration leads to many users adopting a product, continuing use depends on its developers understanding how the software is used and new features that users may want.

Apart from inspiration, there are three factors that drive the design of software products:

1. **Business and consumer needs that are not met by current products** For example, book and magazine publishers are moving to both online and paper publication, yet few software products allow seamless conversion from one medium to another.
2. **Dissatisfaction with existing business or consumer software products** For example, many current software products are bloated with features that are rarely used. New companies may decide to produce simpler products in the same area that meet the needs of most users.

3. **Changes in technology that make completely new types of products possible** For example, as virtual reality (VR) technology matures and the hardware gets cheaper, new products may exploit this opportunity.

As I explained in **Chapter 1**, software products are not developed to meet the requirements of a specific client. Consequently, techniques that have been developed for eliciting, documenting, and managing software requirements aren't used for product engineering. You don't need to have a complete and detailed requirements document as part of a software development contract. There is no need for prolonged consultations when requirements change. Product development is incremental and agile, so you can use less formal ways of defining your product.

In the early stage of product development, rather than understanding the requirements of a specific client, you are trying to understand what product features will be useful to users and what they like and dislike about the products that they use. Briefly, a feature is a fragment of functionality, such as a Print feature, a Change Background feature, a New Document feature, and so on. Before you start programming a product, you should create a list of features to be included in your product. This is your starting point for product design and development.

It makes sense in any product development to spend time trying to understand the potential users and customers of your product. A range of techniques have been developed for understanding the ways that people work and use software. These include user interviews, surveys, ethnography, and task analysis.[1] Some of these techniques are expensive and unrealistic for small companies. However, informal user analysis and discussions, which simply involve asking users about their work, the software that they use, and its strengths and weaknesses, are inexpensive and very valuable.

[1] I discuss techniques of user analysis to discover software requirements in my general software engineering textbook, *Software Engineering*, 10th edition

(Pearson Education, 2015).

One problem with informal user studies for business products is that the users simply may not want new software. For business products, the business buys the product, but its employees are the users. These users may be hostile to new products because they have to change their familiar way of working or perhaps because increased automation may reduce the number of jobs available. Business managers may suggest what they want from a new software product, but this does not always reflect the needs or wishes of the product's users.

In this chapter, I assume that informal user consultations are possible. I explain ways of representing users (personas) and communicating with them and other product stakeholders. I focus on how short, natural language descriptions (scenarios and stories) can be used to visualize and document how users might interact with a software product.

Figure 3.1 shows that personas, scenarios, and user stories lead to features that might be implemented in a software product.

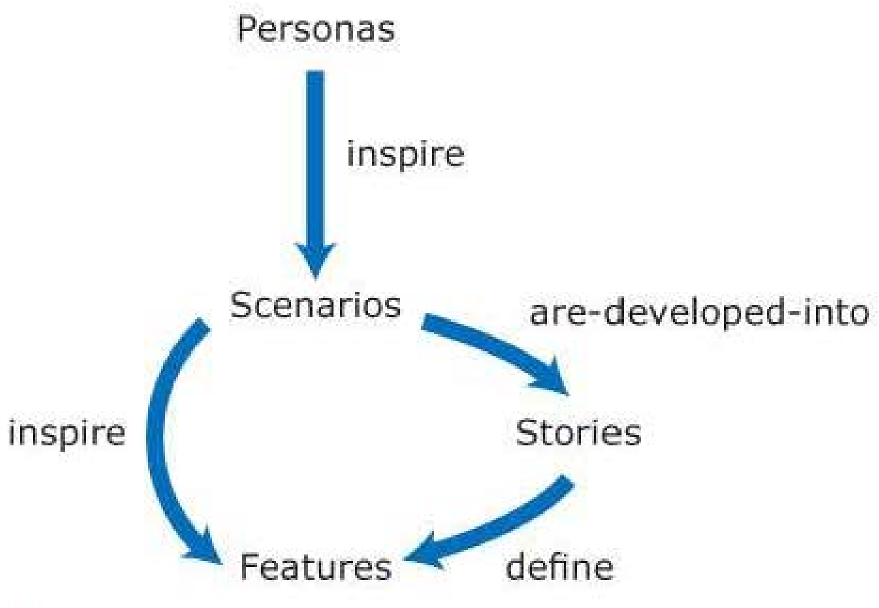


Figure 3.1
From personas to features

If you look on the web, you can find a range of definitions of a "product feature," but I think of a feature as a fragment of functionality that implements some user or system need. You access features through the user interface of a product. For example, the editor that I used to write this book includes a feature to create a "New Group," in which a group is a set of documents that is accessed as a pull-down menu.

A feature is something that the user needs or wants. You can write a user story to make this explicit:

As an author I need a way to organize the text of the book that I'm writing into chapters and sections.

Using the New Group feature, you create a group for each chapter, and the documents within that group are the sections of that chapter. This feature can be described using a short, narrative, feature description:

The "New Group" command, activated by a menu choice or keyboard shortcut, creates a named container for a set of documents and groups.

Alternatively, you can use a standard template where you define the feature by its input, its functionality, its output, and how it is activated. **Figure 3.2** shows the elements of the standard template, and **Figure 3.3** shows how the New Group feature can be defined using this template.

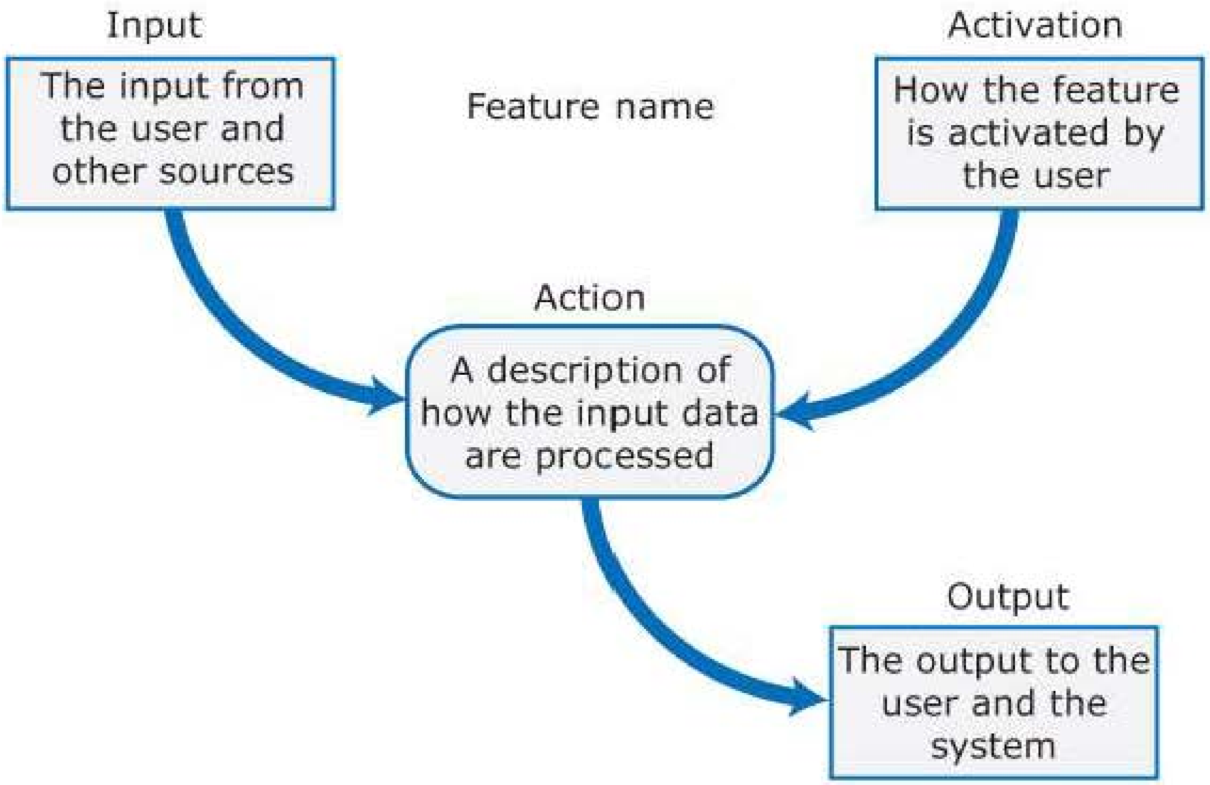

Input
The input from the user and other sources
Feature name
Activation
How the feature is activated by the user
Action
A description of how the input data are processed
Output
The output to the user and the system

Figure 3.2
Feature description

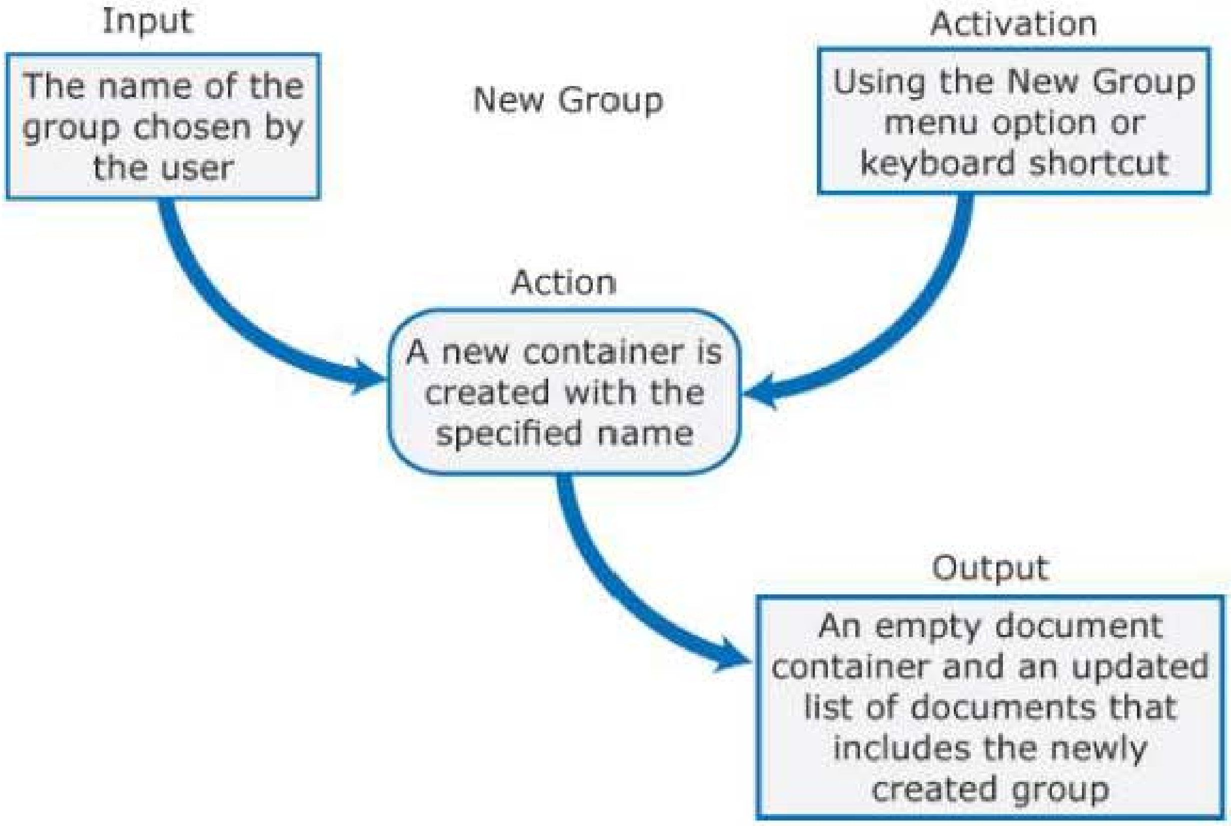

Input
The name of the group chosen by the user
New Group
Activation
Using the New Group menu option or keyboard shortcut
Action
A new container is created with the specified name
Output
An empty document container and an updated list of documents that includes the newly created group

Figure 3.3
The New Group feature description

Features can be defined using the input/action/output model that I have shown. However, system developers often need only a short feature description; then they fill in the feature details. This is especially true when the feature is a "utility" feature that is commonly available in other products. For example, the Cut and Paste feature is well known, so might be defined as:

Cut and Paste – any selected object can be cut or copied, then inserted elsewhere into the document.

Sometimes all you need to say is that a Cut and Paste feature should be included and then rely on the developer's general understanding to implement this.

Features are the fundamental elements of an agile method called Feature-Driven Development (FDD). I have no experience with this method and have not met anyone who uses it for product development, so I can't comment on it directly. One aspect of this approach that I have used, however, is its template for feature description:

<*action*> **the** <*result*> <**by|for|of|to**> <*object*>

So, the New Group feature above could be described as:

Create **the** *container* **for** *documents or groups*

I show another example of this simple approach to feature description in **Section 3.4.2**.

I return to the topic of features in **Section 3.4** after I have described scenarios and user stories. You can use both of these narrative techniques for deriving the list of features to be included in a system.

3.1 Personas

One of the first questions you should ask when developing a software product is "Who are the target users for my product?" You need to have some understanding of your potential users in order to design features that they are likely to find useful and to design a user interface that is suited to them.

Sometimes you know the users. If you are a software engineer developing software tools for other engineers, then you understand, to some extent at least, what they are likely to want. If you are designing a phone or tablet app for general use, you may talk to friends and family to understand what potential users may like or dislike. In those circumstances, you may be able to design using an intuitive idea of who the users are and what they can do. However, you should be aware that your users are diverse and your own experience may not provide a complete picture of what they might want and how they might work.

For some types of software products, you may not know much about the background, skills, and experience of potential users. Individuals on the development team may have their own ideas about the product users and their capabilities. This can lead to product inconsistencies as these different views are reflected in the software implementation. Ideally, the team should have a shared vision of users, their skills, and their motivations for using the software. Personas are one way of representing this shared vision.

Personas are about "imagined users," character portraits of types of user that you think might adopt your product. For example, if your product is aimed at managing appointments for dentists, you might create a dentist persona, a receptionist persona, and a patient persona. Personas of different types of users help you imagine what these users may want to do with your software and how they might use it. They also help you envisage difficulties that users might have in understanding and using product features.

A persona should paint a picture of a type of product user. You should describe the users' backgrounds and why they might want to use your product. You should also say something about their education and technical skills. This helps you assess whether or not a software feature is likely to be useful and understandable by typical product users.

An example of a persona that I wrote when designing the iLearn system, described in Chapter 1, is shown in Table 3.1. This is the persona of a teacher who is committed to digital learning and who believes that using digital devices can enhance the overall learning process.

Table 3.1 A persona for a primary school teacher

Jack, a primary school teacher
Jack, age 32, is a primary school (elementary school) teacher in Ullapool, a large coastal village in the Scottish Highlands. He teaches children from ages 9 to 12. He was born in a fishing community north of Ullapool, where his father runs a marine fuels supply business and his mother is a community nurse. He has a degree in English from Glasgow University and retrained as a teacher after several years working as a web content author for a large leisure group.
Jack's experience as a web developer means that he is confident in all aspects of digital technology. He passionately believes that the effective use of digital technologies, blended with face-to-face teaching, can enhance the learning experience for children. He is particularly interested in using the iLearn system for project-based teaching, where students work together across subject areas on a challenging topic.

There is no standard way of representing a persona; if you search the web, you will find a number of different recommendations. Common features of these suggestions are shown in Figure 3.4. The recommended aspects of a persona description—namely, personalization, relevance, education, and job-related—are explained in **Table 3.2**.

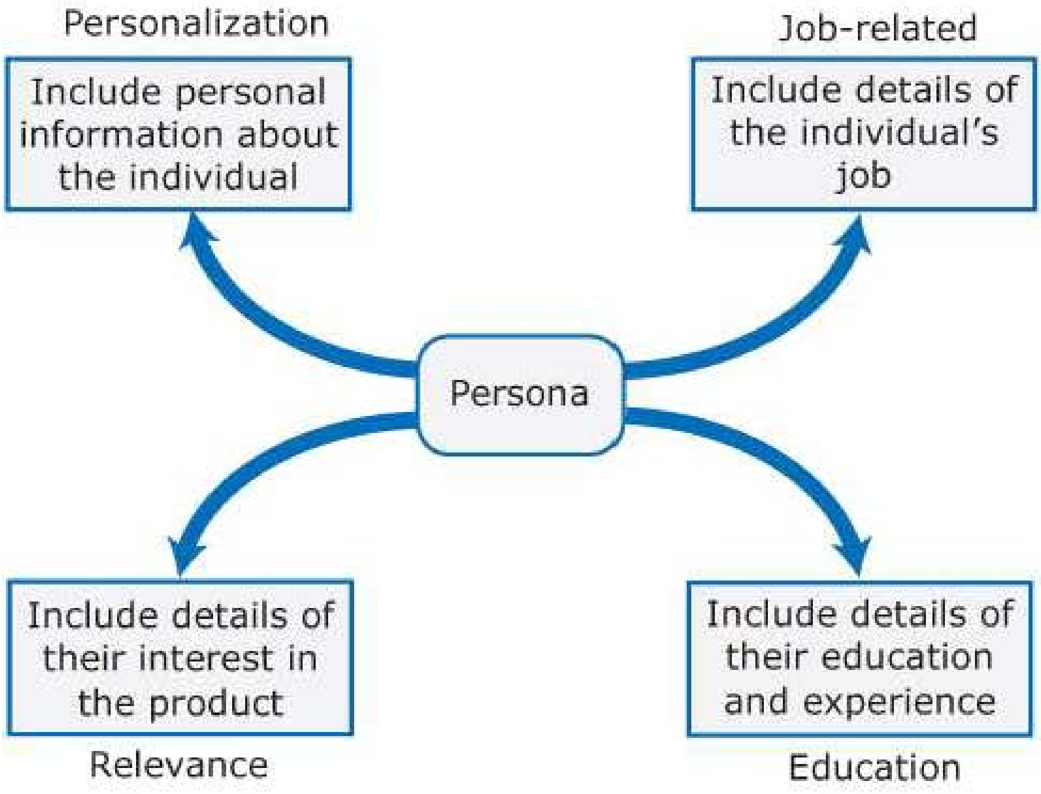

Personalization
Include personal information about the individual
Job-related
Include details of the individual's job
Persona
Include details of their interest in the product
Relevance
Include details of their education and experience
Education

Figure 3.4
Persona descriptions

Table 3.2 Aspects of persona descriptions

Aspect	Description
Personalization	You should give them a name and say something about their personal circu mstances. It is sometimes helpful to use an appropriate stock photograph to represent the person in the person a. Some studies suggest that this hel ps project teams use personas more e ffectively.
Job-related	If your product is targeted at busines s, you should say something about th

	eir job and (if necessary) what that jo b involves. For some jobs, such as a t eacher where readers are likely to be familiar with the job, this may not be necessary.
Education	You should describe their educationa l background and their level of techni cal skills and experience. This is imp ortant, especially for interface desig n.
Relevance	If you can, you should say why they might be interested in using the prod uct and what they might want to do with it.

Many recommendations about what should be included in a persona suggest describing the individual's goals. I disagree because it's impossible to pin down what is meant by "goals." Goals are a broad concept. Some people have goals of self-improvement and learning; others have work-oriented goals of career progression and promotion. For some people, goals are not related to work. Their goals are simply to get through the day and earn enough so that they can do the things outside of work that give them pleasure and satisfaction.

My experience is that trying to define "user goals" is not helpful. I have found that most people don't have clearly defined goals when they use software. They may have been told to use the software as part of their job, they may see the software as a way to do their work more effectively, or they may find the software useful in organizing their life. Rather than try to set out goals, I think it's better to try to explain why the software might be useful and to give examples of what potential users may want to do with it.

If your product is targeted at a specific group of users, you may need only one or two personas to represent potential system users. For some products, however, the user group may be very broad and you may think that a large number of personas are needed. In fact, using many personas can make it more difficult to design a coherent system because they inevitably overlap. In general, you don't need more than five personas to help identify the key features of a system.

Personas should be relatively short and easy to read. For the iLearn system, we developed personas that we described in two or three paragraphs of text. We found that these had enough information to be useful. Two of the personas that we created are shown in **Tables 3.3** and **3.4**.

Table 3.3 A persona for a history teacher

Emma, a history teacher
Emma, age 41, is a history teacher in a secondary school (high school) in Edinburgh. She teaches students from ages 12 to 18. She was born in Cardiff in Wales, where both her father and her mother were teachers. After completing a degree in history from Newcastle University, she moved to Edinburgh to be with her partner and trained as a teacher. She has two children, aged 6 and 8, who both attend the local primary school. She likes to get home as early as she can to see her children, so often does lesson preparation, administration, and marking from home. Emma uses social media and the usual productivity applications to prepare her lessons, but is not particularly interested in digital technologies. She hates the virtual learning environment that is currently used in her school and avoids using it if she can. She believes that face-to-face teaching is most effective. She might use the iLearn system for administration and access to historical films and documents. However, she is not interested in a blended digital/face-to-face approach to teaching.

Table 3.4 A persona for an IT technician

Elena, a school IT technician
Elena, age 28, is a senior IT technician in a large secondary school (high school) in Glasgow with over 2000 students. Originally from Poland, she has a diploma in electronics from Potsdam University. She moved to Scotland in 2011 after being unemployed for a year after graduation. She has a Scottish partner, no children, and hopes to develop her career in Scotland. She was originally appointed as a junior technician but was promoted, in 2014, to a senior post responsible for all the school computers.
Although not involved directly in teaching, Elena is often called on to help in computer science classes. She is a competent Python programmer and is a "power user" of digital technologies. She has a long-term career goal of becoming a technical expert in digital learning technologies and being involved in their development. She wants to become an expert in the iLearn system and sees it as an experimental platform for supporting new uses for digital learning.

The persona in **Table 3.3** represents users who do not have a technical background. They simply want a system to provide support for administration. Elena's persona in **Table 3.4** represents technically skilled support staff who may be responsible for setting up and configuring the iLearn software.

I haven't included personas for the students who were intended to be the ultimate end-users of the iLearn system. The reason is that we saw the iLearn system as a platform product that should be configured to suit the preferences and needs of individual schools and teachers. Students would use iLearn to access tools. However, they would not use the configuration features that make iLearn a unique system. Although we planned to include a standard set of applications with the system, we were driven by the belief that the best people to create learning systems were teachers, supported by local technical staff.

Ideally, software development teams should be diverse, with people of different ages and genders. However, the reality is that software product developers are still, overwhelmingly, young men with a high level of technical skill. Software users are more diverse, with varying levels of technical skill. Some developers find it hard to appreciate the problems that users may have with the software. An important benefit of personas is that they help the development team members empathize with potential users of the software. Personas are a tool that allows team members to "step into the users' shoes." Instead of thinking about what they would do in a particular situation, they can imagine how a persona would behave and react.

So, when you have an idea for a feature, you can ask "Would this persona be interested in this feature?" and "How would that persona access and use the feature?". Personas can help you check your ideas to make sure that you are not including product features that aren't really needed. They help you to avoid making unwarranted assumptions, based on your own knowledge, and designing an overcomplicated or irrelevant product.

Personas should be based on an understanding of the potential product users: their jobs, their backgrounds, and their aspirations. You should study and survey potential users to understand what they want and how they might use the product. From these data, you can abstract the essential information about the different types of product users and then use this as a basis for creating personas. These personas should then be cross-checked against the user data to make sure that they reflect typical product users.

It may be possible to study the users when your product is a development of an existing product. For new products, however, you may find it impractical to carry out detailed user surveys. You may not have easy access to potential users. You may not have the resources to carry out user surveys, and you may want to keep your product confidential until it is ready to launch.

If you know nothing about an area, you can't develop reliable personas, so you need to do some user research before you start. This does not have to be a formal or prolonged process. You may know people working in an area and they might be able to help you meet their colleagues to discuss your ideas. For example, my daughter is a teacher and she helped arrange lunch with a group of her colleagues to discuss how they use digital technologies. This helped with both persona and scenario development.

Personas that are developed on the basis of limited user information are called proto-personas. Proto-personas may be created as a collective team exercise using whatever information is available about potential product users. They can never be as accurate as personas developed from detailed user studies, but they are better than nothing. They represent the product users as seen by the development team, and they allow the developers to build a common understanding of the potential product users.

3.2 Scenarios

As a product developer, your aim should be to discover the product features that will tempt users to adopt your product rather than competing software. There is no easy way to define the "best" set of product features. You have to use your own judgement about what to include in your product. To help select and design features, I recommend that you invent scenarios to imagine how users could interact with the product that you are designing.

A scenario is a narrative that describes a situation in which a user is using your product's features to do something that they want to do. The scenario should briefly explain the user's problem and present an imagined way that the problem might be solved. There is no need to include everything in the scenario; it isn't a detailed system specification.

Table 3.5 is an example scenario that shows how Jack, whose persona I described in **Section 3.2**, might use the iLearn system.

Table 3.5 Jack's scenario: Using the iLearn system for class projects

Fishing in Ullapool
Jack is a primary school teacher in Ullapool, teaching P6 pupils. He has decided that a class project should be focused around the fishing industry in the area, looking at the history, development, and economic impact of fishing. As part of this, students are asked to gather and share reminiscences from relatives, use newspaper archives, and collect old photographs related to fishing and fishing communities in the area. Pupils use an iLearn wiki to gather together fishing stories and SCRAN (a history archive site) to access newspaper archives and photographs. However, Jack also needs a photo-sharing site as he

wants students to take and comment on each others' photos and to upload sc
ans of old photographs that they may have in their families. He needs to be ab
le to moderate posts with photos before they are shared, because pre-teen chi
ldren can't understand copyright and privacy issues.

Jack sends an email to a primary school teachers' group to see if anyone can r
ecommend an appropriate system. Two teachers reply and both suggest that
he use KidsTakePics, a photo-sharing site that allows teachers to check and
moderate content. As KidsTakePics is not integrated with the iLearn authenti
cation service, he sets up a teacher and a class account with KidsTakePics.

He uses the the iLearn setup service to add KidsTakePics to the services seen b
y the students in his class so that, when they log in, they can immediately use
the system to upload photos from their phones and class computers.

From this description of how iLearn might be used for class projects, you can see some of the key elements of a scenario (**Figure 3.5**) that may be included to help you think about the product features that you need.

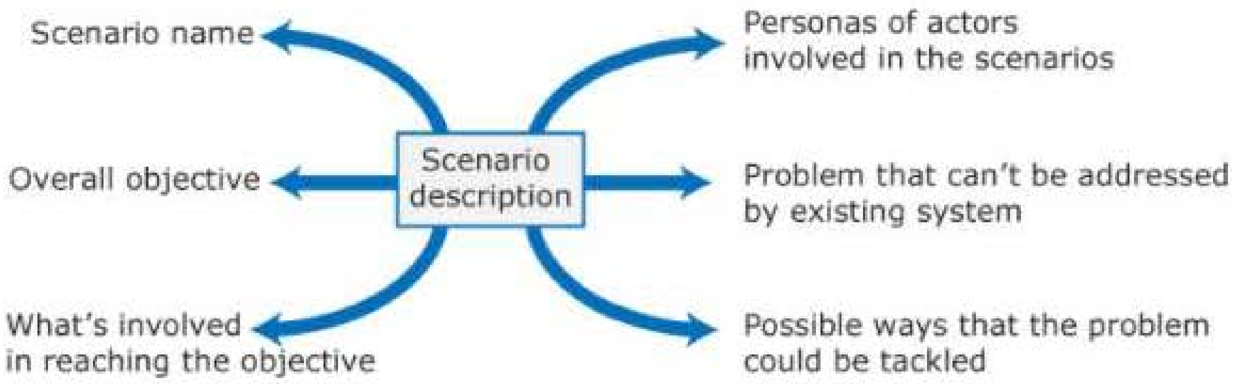

Scenario name
Personas of actors involved in the scenarios
Scenario description
Overall objective
Problem that can't be addressed by existing system
What's involved in reaching the objective
Possible ways that the problem could be tackled

Figure 3.5
Elements of a scenario description

The most important elements of a scenario are:

1. A brief statement of the overall objective. In Jack's scenario, shown in **Table 3.5**, this is to support a class project on the fishing industry.

2. References to the persona involved (Jack) so that you can get information about the capabilities and motivation of that user.
3. Information about what is involved in doing the activity. For example, in Jack's scenario, this involves gathering reminiscences from relatives, accessing newspaper archives, and so on.
4. If appropriate, an explanation of problems that can't be readily addressed using the existing system. Young children don't understand issues such as copyright and privacy, so photo sharing requires a site that a teacher can moderate to make sure that published images are legal and acceptable.
5. A description of one way that the identified problem might be addressed. This may not always be included especially if technical knowledge is needed to solve the problem. In Jack's scenario, the preferred approach is to use an external tool designed for school students.

The idea of using scenarios to support software engineering has been around since the 1980s. Different types of scenarios have been proposed, ranging from high-level scenarios like this one about Jack to more detailed and specific scenarios that set out the steps involved in a user's interaction with a system. They are the basis for both use cases, which are extensively used in object-oriented methods, and user stories, which are used in agile methods. Scenarios are used in the design of requirements and system features, in system testing, and in user interface design.

Narrative, high-level scenarios, like Jack's scenario, are primarily a means of facilitating communication and stimulating design creativity. They are effective in communication because they are understandable and accessible to users and to people responsible for funding and buying the system.[2] Like personas, they help developers to gain a shared understanding of the system that they are creating. You should always be aware, however, that scenarios are not specifications. They lack detail, they may be incomplete, and they may not represent all types of user interactions.

2I presented some of the scenarios for the iLearn system to a government minister for education. He commented that this was the first time he had ever attended a meeting on an IT system where he actually understood what the system was supposed to do.

Some people recommend that scenarios should be structured with different fields such as what the user sees at the beginning of a scenario, a description of the normal flow of events, a description of what might go wrong, and so on. If you are using scenarios to elicit detailed requirements, the benefit of structure is that you have a consistent presentation, which means you are less likely to forget to include elements that relate to critical system requirements. Engineers usually like this structured approach. However, my experience is that system users, who read and check the scenarios, find structured scenarios to be intimidating and hard to understand.

Consequently, when you are using scenarios at the early stages of product design, I recommend narrative rather than structured scenarios. These scenarios may range from two or three paragraphs of text, like Jack's scenario in **Table 3.5**, to longer descriptions. You may need to write longer descriptions when your software will be used in existing processes and must interoperate with other software. Your scenarios may then have to include descriptions of interactions with other processes and software systems.

Emma's scenario (shown in **Table 3.6**) is an example of a longer scenario. In it, she uses the iLearn system to help her with the administration involved in setting up a class trip.

Table 3.6 Using the iLearn system for administration

Emma is teaching the history of World War I to a class of 14-year-olds (S3). A group of S3 students are visiting the historic World War I battlefields in north ern France. She wants to set up a "battlefields group" where the students who are attending the trip can share their research about the places they are visiti ng as well as their pictures and thoughts about the visit.

From home, she logs onto the iLearn system using her Google account credentials. Emma has two iLearn accounts— her teacher account and a parent account associated with the local primary school. The system recognizes that she is a multiple account owner and asks her to select the account to be used. She chooses the teacher account and the system generates her personal welcome screen. As well as her selected applications, this also shows management apps that help teachers create and manage student groups.

Emma selects the "group management" app, which recognizes her role and school from her identity information and creates a new group. The system prompts for the class year (S3) and subject (history) and automatically populates the new group with all S3 students who are studying history. She selects those students going on the trip and adds her teacher colleagues, Jamie and Claire, to the group.

She names the group and confirms that it should be created. The app sets up an icon on her iLearn screen to represent the group, creates an email alias for the group, and asks Emma if she wishes to share the group. She shares access with everyone in the group, which means that they also see the icon on their screen. To avoid getting too many emails from students, she restricts sharing of the email alias to Jamie and Claire.

The group management app then asks Emma if she wishes to set up a group web page, wiki, and blog. Emma confirms that a web page should be created and she types some text to be included on that page.

She then accesses Flickr using the icon on her screen, logs in, and creates a private group to share trip photos that students and teachers have taken. She uploads some of her own photos from previous trips and emails an invitation to join the photo-sharing group to the battlefields email list. Emma uploads material from her own laptop that she has written about the trip to iLearn and shares this with the battlefields group. This action adds her documents to the web page and generates an alert to group members that new material is available.

Emma's scenario is different from Jack's scenario because it describes a common and well-understood process rather than something new. The scenario discusses how parts of the process (setting up an email group and web page) are automated by the iLearn system. Remember that Emma is an e-learning skeptic; she is not interested in innovative applications. She wants a system that will make her life easier and reduce the amount of routine administration that she has to do.

In this type of scenario, you are much more likely to include specific details of what might be in the system. For example, it explains that Emma logs in to the system with her Google credentials. This means she doesn't have to remember a separate login name and password. I discuss this approach to authentication in **Chapter 7**.

When you see this kind of information in a scenario, you need to check whether this is what the user really needs or whether it represents a more general requirement. The statement that the software has to support login using Google credentials might actually reflect a more general need—to provide a login mechanism in which users don't have to remember yet another set of credentials. You may decide on an alternative approach to authentication, such as fingerprint or face recognition on a mobile phone, that avoids the need for system-specific login credentials.

3.2.1 Writing scenarios

Your starting point for scenario writing should be the personas you have created. You should normally try to imagine several scenarios for each persona. Remember that these scenarios are intended to stimulate thinking rather than provide a complete description of the system. You don't need to cover everything you think users might do with your product.

Scenarios should always be written from the user's perspective and should be based on identified personas or real users. Some writers suggest that scenarios should focus on goals—what the user wants to do

—rather than the mechanisms they use to do this. They argue that scenarios should not include specific details of an interaction as these limit the freedom of feature designers. I disagree. As you can see from Emma's scenario, it sometimes makes sense to talk about mechanisms, such as login with Google, that both product users and developers understand.

Furthermore, writing scenarios in a general way that doesn't make assumptions about implementation can be potentially confusing for both users and developers. For example, I think that "X cuts paragraphs from the newspaper archive and pastes them into the project wiki" is easier to read and understand than "X uses an information transfer mechanism to move paragraphs from the newspaper archive to the project wiki."

Sometimes there may be a specific requirement to include a particular feature in the system because that feature is widely used. For example, Jack's scenario in **Table 3.5** discusses the use of an iLearn wiki. Many teachers currently use wikis to support group writing and they specifically want to have wikis in the new system. Such user requirements might be even more specific. For example, when designing the iLearn system, we discovered that teachers wanted Wordpress blogs, not just a general blogging facility. So, you should include specific details in a scenario when they reflect reality.

Scenario writing is not a systematic process and different teams approach it in different ways. I recommend that each team member be given individual responsibility for creating a small number of scenarios and work individually to do this. Obviously, members may discuss the scenarios with users and other experts, but this is not essential. The team then discusses the proposed scenarios. Each scenario is refined based on that discussion.

Because it is easy for anyone to read and understand scenarios, it is possible to get users involved in their development. For the iLearn system, we found that the best approach was to develop an imaginary scenario based on our understanding of how the system might be used and then ask users to tell us what we got wrong. Users could ask about things they did not understand, such as "Why can't a photo-sharing site like Flickr be used in Jack's scenario?" They could suggest how the scenario could be extended and made more realistic.

We tried an experiment in which we asked a group of users to write their own scenarios about how they might use the system. This was not a success. The scenarios they created were simply based on how they worked at the moment. They were far too detailed and the writers couldn't easily generalize their experience. Their scenarios were not useful because we wanted something to help us generate ideas rather than replicate the systems that they already used.

Scenarios are not software specifications but are ways of helping people think about what a software system should do. There is no simple answer to the question "How many scenarios do I need?" In the iLearn system, 22 scenarios were developed to cover different aspects of system use. There was quite a lot of overlap among these scenarios, so it was probably more than we really needed. Generally, I recommend developing three or four scenarios per persona to get a useful set of information.

Although scenarios certainly don't have to describe every possible use of a system, it is important that you look at the roles of each of the personas that you have developed and write scenarios that cover the main responsibilities for that role. Jack's scenario and Emma's scenario are based on using the iLearn system to support teaching.

Like other system products designed for use in an organization, however, iLearn needs to be configured for use. While some of this configuration can be done by tech-savvy teachers, in many schools it is technical support staff who have this responsibility. Elena's scenario, shown in **Table 3.7**, describes how she might configure the iLearn software.

Table 3.7 Elena's scenario: Configuring the iLearn system

Elena has been asked by David, the head of the art department in her school, to help set up an iLearn environment for his department. David wants an environment that includes tools for making and sharing art, access to external websites to study artworks, and "exhibition" facilities so that the students' work can be displayed.

Elena starts by talking to art teachers to discover the tools that they recommend and the art sites that they use for studies. She also discovers that the tools they use and the sites they access vary according to the age of their students. Consequently, different student groups should be presented with a toolset that is appropriate for their age and experience.

Once she has established what is required, Elena logs into the iLearn system as an administrator and starts configuring the art environment using the iLearn setup service. She creates sub-environments for three age groups plus a shared environment that includes tools and sites that may be used by all students.

She drags and drops tools that are available locally and the URLs of external websites into each of these environments. For each of the sub-environments, she assigns an art teacher as its administrator so that they can't refine the tool and website selection that has been set up. She publishes the environments in "review mode" and makes them available to the teachers in the art department.

After discussing the environments with the teachers, Elena shows them how to refine and extend the environments. Once they have agreed that the art environment is useful, it is released to all students in the school.

Writing scenarios always gives you ideas for the features that you can include in the system. You may then develop these ideas in more detail by analyzing the text of the scenario, as I explain in the next section.

3.3 User stories

I explained in Section 3.2 that scenarios are descriptions of situations in which a user is trying to do something with a software system. Scenarios are high-level stories of system use. They should describe a sequence of interactions with the system but should not include details of these interactions.

User stories are finer-grain narratives that set out in a more detailed and structured way a single thing that a user wants from a software system. I presented a user story at the beginning of the chapter:

As an author I need a way to organize the book that I'm writing into chapters and sections.

This story reflects what has become the standard format of a user story:

As a <role>, I <want / need> to <do something>

Another example of a user story taken from Emma's scenario might be:

As a teacher, I want to tell all members of my group when new information is available.

A variant of this standard format adds a justification for the action:

As a <role> I <want / need> to <do something> so that <reason>

For example:

As a teacher, I need to be able to report who is attending a class trip so that the school maintains the required health and safety records.

Some people argue that a rationale or justification should always be part of a user story. I think this is unnecessary if the story makes sense on its own. Knowing one reason why this might be useful doesn't help the product developers. However, in situations where some developers are unfamiliar with what users do, a rationale can help those developers understand why the story has been included. A rationale may also help trigger ideas about alternative ways of providing what the user wants.

An important use of user stories is in planning, and many users of the Scrum method represent the product backlog as a set of user stories. For this purpose, user stories should focus on a clearly defined system feature or aspect of a feature that can be implemented within a single sprint. If the story is about a more complex feature that might take several sprints to implement, then it is called an "epic." An example of an epic might be:

As a system manager, I need a way to back up the system and restore individual applications, files, directories, or the whole system.

A lot of functionality is associated with this user story. For implementation, it should be broken down into simpler stories, with each story focusing on a single aspect of the backup system.

When you are thinking about product features, user stories are not intended for planning but for helping with feature identification. Therefore, you don't need to be overly concerned about whether your stories are simple stories or epics. You should aim to develop stories that are helpful in one of two ways:

- as a way of extending and adding detail to a scenario;
- as part of the description of the system feature that you have identified.

As an example of scenario refinement, the initial actions in Emma's scenario shown in **Figure 3.6** can be represented by three user stories. Recall that the scenario says:

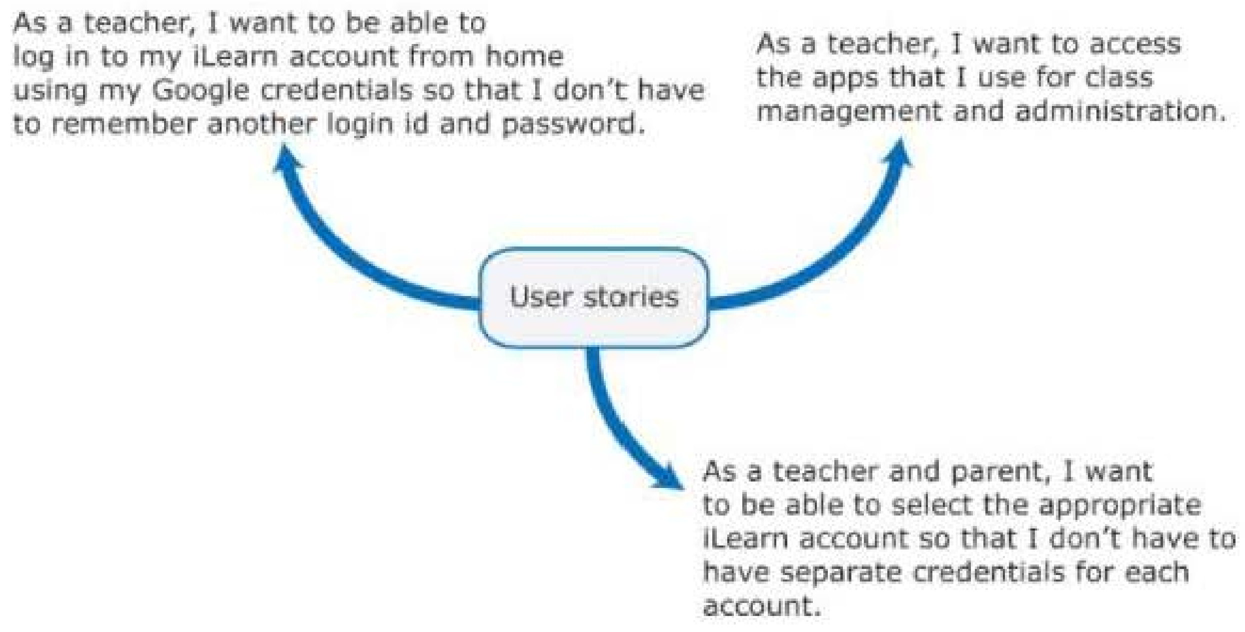

As a teacher, I want to be able to log in to my iLearn account from home using my Google credentials so that I don't have to remember another login id and password.
As a teacher, I want to access the apps that I use for class management and administration.
User stories
As a teacher and parent, I want to be able to select the appropriate iLearn account so that I don't have to have separate credentials for each account.

Figure 3.6
User stories from Emma's scenario

From home, she logs onto the iLearn system using her Google account credentials. Emma has two iLearn accounts—her teacher account and a parent account associated with the local primary school. The system recognizes that she is a multiple account owner and asks her to select the account to be used. She chooses the teacher account and the system generates her personal welcome screen. As well as her selected applications, this also shows management apps that help teachers create and manage student groups.

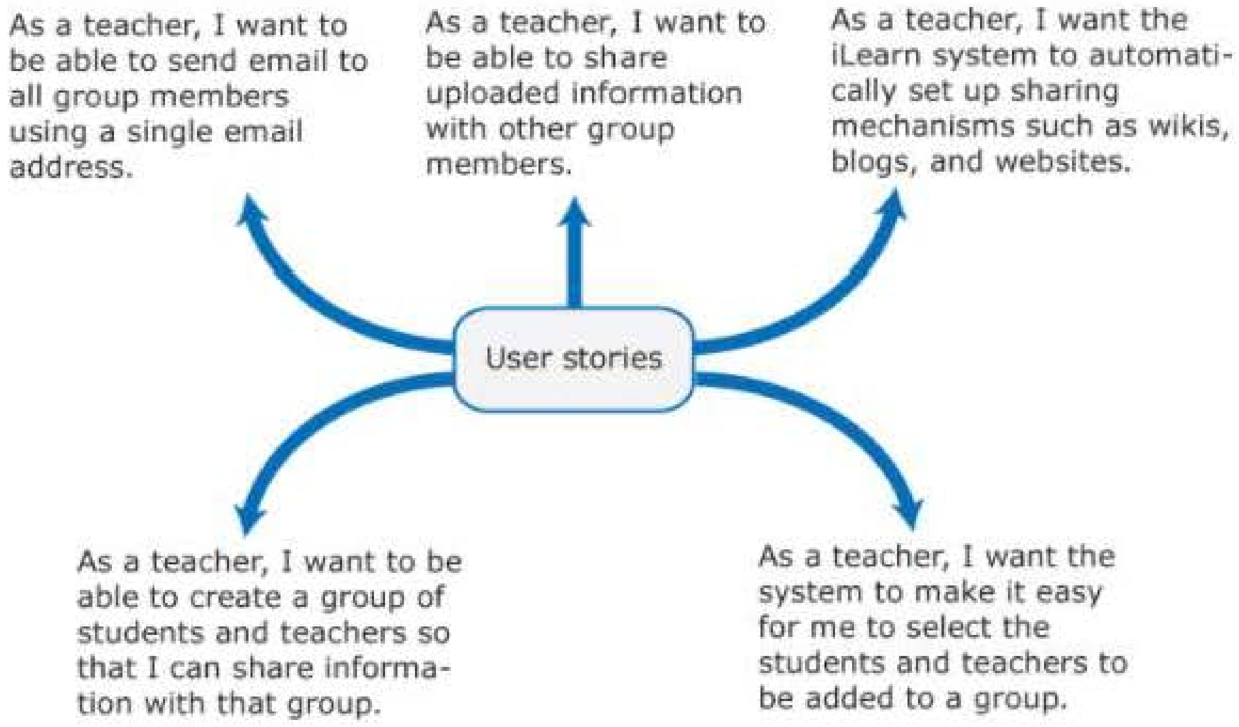
You can create user stories from this account, as I show in **Figure 3.6**.

You can see from the stories in **Figure 3.6** that I have included a rationale in the story that explains why Emma wants to work in the way specified in the scenario. It is best not to include rationale in a scenario, as it tends to disrupt the flow of the description and make it more difficult to read and understand.

When you define user stories from a scenario, you provide more information to developers to help them design the product's features. We can see an example of this in the stories shown in **Figure 3.6**.

Emma wants an easy way to authenticate herself to the system, either as a teacher or as a parent. She doesn't want to have to remember more login credentials, and she doesn't want to have two accounts with different credentials.

As an example of how stories can be used to describe the features of a system, Emma's scenario discusses how a group can be created and explains system actions that take place on group creation. The user stories shown in **Figure 3.7** can be derived from the scenario to describe the Groups feature in the iLearn system.



As a teacher, I want to be able to send email to all group members using a single email address.
As a teacher, I want to be able to share uploaded information with other group members.
As a teacher, I want the iLearn system to automatically set up sharing mechanisms such as wikis, blogs, and websites.
User stories
As a teacher, I want to be able to create a group of students and teachers so that I can share information with that group.
As a teacher, I want the system to make it easy for me to select the students and teachers to be added to a group.

Figure 3.7
User stories describing the Groups feature

The set of stories shown in **Figure 3.7** is not a complete description of the Groups feature. No stories are concerned with deleting or changing a group, restricting access, and other tasks. You start by deriving stories from a scenario, but you then have to think about what other stories might be needed for a complete description of a feature's functionality.

A question that is sometimes asked about user stories is whether you should write "negative stories" that describe what a user doesn't want. For example, you might write this negative story:

As a user, I don't want the system to log and transmit my information to any external servers.

If you are writing stories to be part of a product backlog, you should avoid negative stories. It is impossible to write system tests that demonstrate a negative. In the early stages of product design, however, it may be helpful to write negative stories if they define an absolute constraint on the system. Alternatively, you can sometimes reframe negative stories in a positive way. For example, instead of the above story, you could write:

As a user, I want to be able to control the information that is logged and transmitted by the system to external servers so that I can ensure that my personal information is not shared.

Some of the user stories that you develop will be sufficiently detailed that you can use them directly in planning. You can include them in a product backlog. Sometimes, however, to use stories in planning, you have to refine the stories to relate them more directly to the implementation of the system.

It is possible to express all of the functionality described in a scenario as user stories. So, an obvious question that you might ask is "Why not just develop user stories and forget about scenarios?" Some agile methods rely exclusively on user stories, but I think that scenarios are more natural and are helpful for the following reasons:

1. Scenarios read more naturally because they describe what a user of a system is actually doing with that system. People often find it easier to relate to this specific information rather than to the statement of wants or needs set out in a set of user stories.

2. When you are interviewing real users or checking a scenario with real users, they don't talk in the stylized way that is used in user stories. People relate better to the more natural narrative in scenarios.
3. Scenarios often provide more context—information about what users are trying to do and their normal ways of working. You can do this in user stories, but it means that they are no longer simple statements about the use of a system feature.

Scenarios and stories are helpful in both choosing and designing system features. However, you should think of scenarios and user stories as "tools for thinking" about a system rather than a system specification. Scenarios and stories that are used to stimulate thinking don't have to be complete or consistent, and there are no rules about how many of each you need.

3.4 Feature identification

As I said in the chapter introduction, your aim at this early stage of product design is to create a list of features that define your software product. A feature is a way of allowing users to access and use your product's functionality so that the feature list defines the overall functionality of the system. In this section, I explain how scenarios and stories can be used to help identify product features.

You should, ideally, identify product features that are independent, coherent and relevant:

1. **Independence** A feature should not depend on how other system features are implemented and should not be affected by the order of activation of other features.
2. **Coherence** Features should be linked to a single item of functionality. They should not do more than one thing, and they should never have side effects.
3. **Relevance** System features should reflect the way users normally carry out some task. They should not offer obscure functionality that is rarely required.

There is no definitive method for feature selection and design. Rather, the four important knowledge sources shown in **Figure 3.8** can help with this.

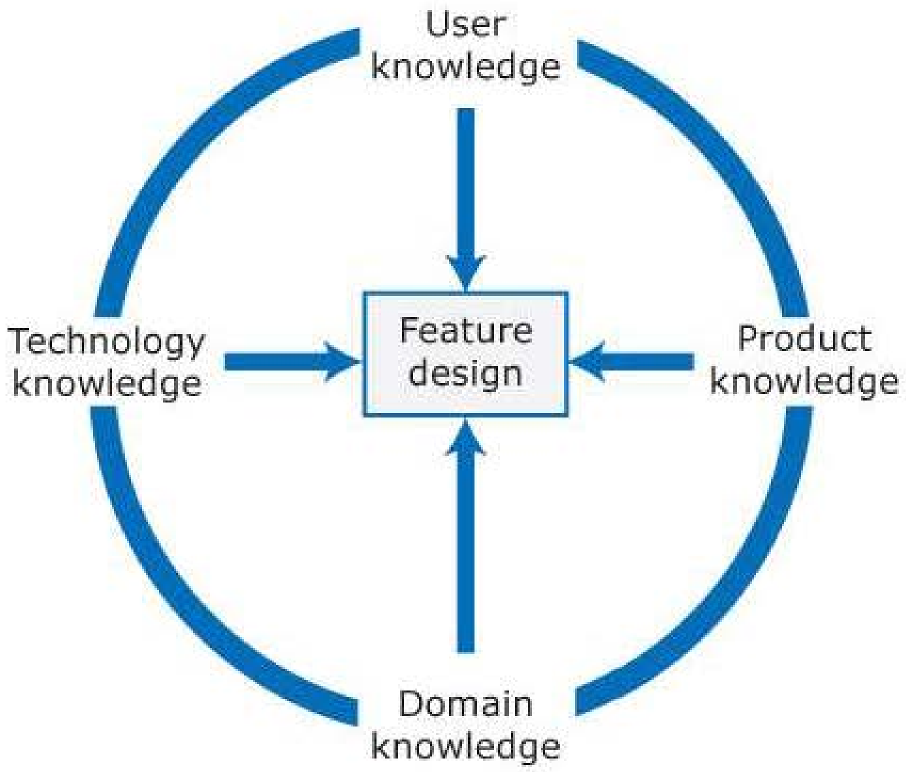


Figure 3.8
Feature design

Table 3.8 explains these knowledge sources in more detail. Of course, these are not all of equal importance for all products. For example, domain knowledge is very important for business products but less important for generic consumer products. You therefore need to think carefully about the knowledge required for your specific product.

Table 3.8 Knowledge required for feature design

Knowledge	Description
User knowledge	You can use user scenarios and user s tories to inform the team of what use rs want and how they might use the s oftware features.
Product knowledge	You may have experience of existing products or decide to research what t hese products do as part of your deve

	lopment process. Sometimes your fe atures have to replicate existing feat ures in these products because they p rovide fundamental functionality tha t is always required.
Domain knowledge	This is knowledge of the domain or w ork area (e.g., finance, event bookin g) that your product aims to support. By understanding the domain, you ca n think of new innovative ways of hel ping users do what they want to do.
Technology knowledge	New products often emerge to take a dvantage of technological developme nts since their competitors were laun ched. If you understand the latest tec hnology, you can design features to make use of it.

Innovation often stems from a combination of domain and technology knowledge. A good example is the Git system for code management that I cover in **Chapter 10**. Git works in a completely different way from previous code management systems. These older systems were based on a technology model in which storage was expensive, so they focused on limiting storage use and delivering information to users as required. The developer of Git realized that storage had become much cheaper, so it was possible for all users to have a complete copy of all information. This allowed for a new approach that dramatically simplified software development by distributed teams.

When you are designing a product feature set and deciding how features should work, you have to consider the six factors shown in **Figure 3.9**.

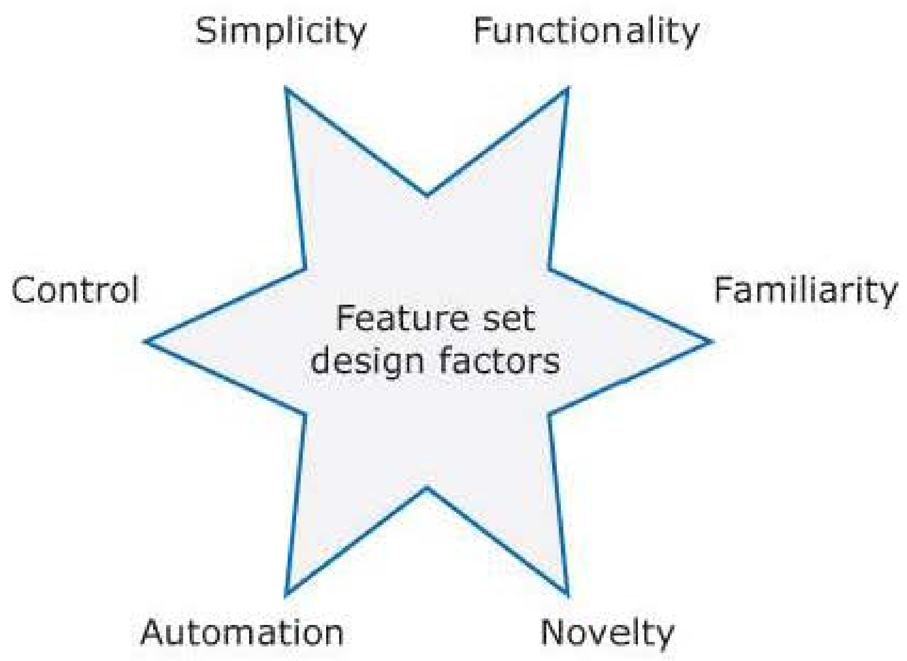


Figure 3.9
Factors in feature set design

Unfortunately, it is impossible to design a feature set in which all of these factors are optimized, so you have to make some trade-offs:

1. **Simplicity and functionality** Everyone says they want software to be as simple as possible to use. At the same time, they demand functionality that helps them do what they want to do. You need to find a balance between providing a simple, easy-to-use system and including enough functionality to attract users with a variety of needs.
2. **Familiarity and novelty** Users prefer that new software should support the familiar everyday tasks that are part of their work or life. However, if you simply replicate the features of a product that they already use, there is no real motivation for them to change. To encourage users to adopt your system, you need to include new features that will convince users that your product can do more than its competitors.
3. **Automation and control** You may decide that your product can automatically do things for users that other products can't. However, users inevitably do things differently from one another. Some may like automation, where the software does things for

them. Others prefer to have control. You therefore have to think carefully about what can be automated, how it is automated, and how users can configure the automation so that the system can be tailored to their preferences.

Your choices have a major influence on the features to be included in your product, how they integrate, and the functionality they provide. You may make a specific choice—for example, to focus on simplicity—that will drive the design of your product.

One problem that product developers should be aware of and try to avoid is "feature creep." Feature creep means that the number of features in a product creeps up as new potential uses of the product are envisaged.

The size and complexity of many large software products such as Microsoft Office and Adobe Photoshop are a consequence of feature creep. Most users use only a relatively small subset of the features of these products. Rather than stepping back and simplifying things, developers continually added new features to the software.

Feature creep adds to the complexity of a product, which means that you are likely to introduce bugs and security vulnerabilities into the software. It also usually makes the user interface more complex. A large feature set often means that you have to bundle vaguely related features together and provide access to these through a higher-level menu. This can be confusing, especially for less experienced users.

Feature creep happens for three reasons:

1. Product managers and marketing executives discuss the functionality they need with a range of different product users. Different users have slightly different needs or may do the same thing but in slightly different ways. There is a natural reluctance to say no to important users, so functionality to meet all of the users' demands ends up in the product.

2. Competitive products are introduced with slightly different functionality to your product. There is marketing pressure to include comparable functionality so that market share is not lost to these competitors. This can lead to "feature wars," where competing products become more and more bloated as they replicate the features of their competitors.
3. The product tries to support both experienced and inexperienced users. Easy ways of implementing common actions are added for inexperienced users and the more complex features to accomplish the same thing are retained because experienced users prefer to work that way.

To avoid feature creep, the product manager and the development team should review all feature proposals and compare new proposals to features that have already been accepted for implementation. The questions shown in **Figure 3.10** may be used to help identify unnecessary features.

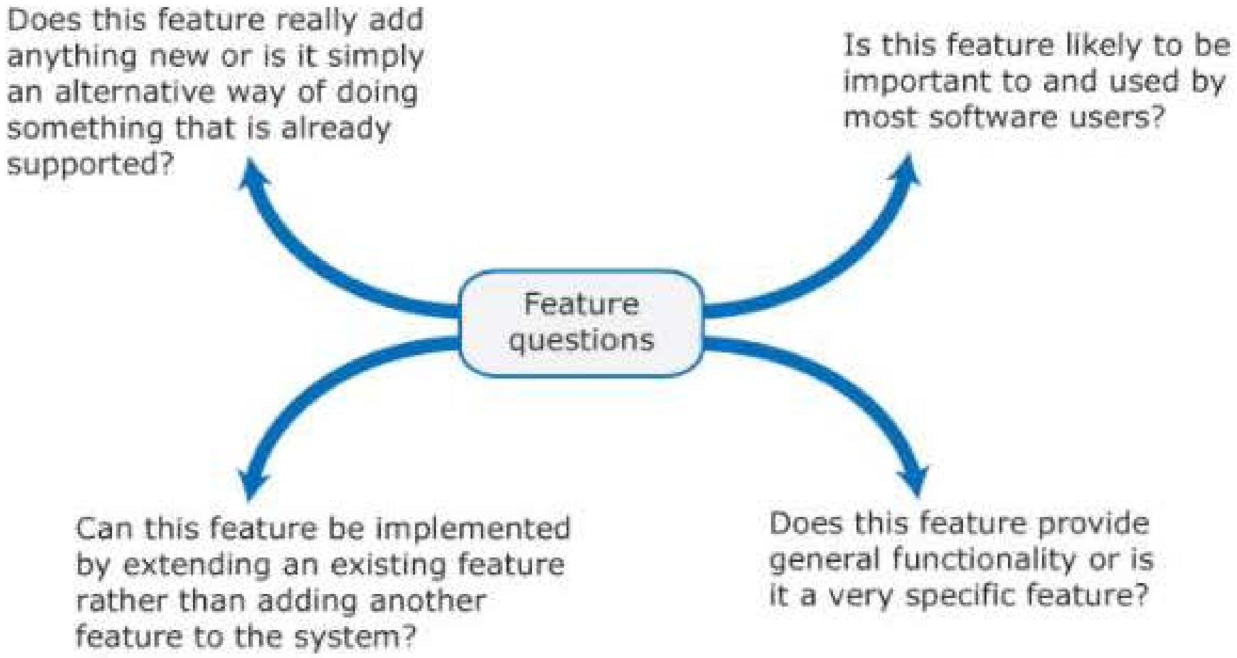

Does this feature really add anything new or is it simply an alternative way of doing something that is already supported?
Is this feature likely to be important to and used by most software users?
Feature questions
Can this feature be implemented by extending an existing feature rather than adding another feature to the system?
Does this feature provide general functionality or is it a very specific feature?

Figure 3.10
Avoiding feature creep

3.4.1 Feature derivation

When you start with a product vision or writing scenarios based on that vision, product features immediately come to mind. I discussed the iLearn system vision in **Chapter 1** and I repeat it in **Table 3.9**.

Table 3.9 The iLearn system vision

FOR teachers and educators WHO need a way to *help students use web-based learning resources and applications,* THE iLearn system is an open learning e nvironment THAT *allows the set of resources used by classes and students to be easily configured for these students and classes by teachers themselves.*
UNLIKE Virtual Learning Environments, such as Moodle, the focus of iLearn i s the learning process rather than the administration and management of ma terials, assessments, and coursework. OUR product *enables teachers to create subject and age-specific environments for their students* using any web-base d resources, such as videos, simulations, and written materials that are appro priate

I have highlighted phrases in this vision suggesting features that should be part of the product, including:

- a feature that allows users to access and use existing web-based resources;
- a feature that allows the system to exist in multiple different configurations;
- a feature that allows user configuration of the system to create a specific environment.

These features distinguish the iLearn system from existing VLEs and are the central features of the product.

This approach of highlighting phrases in a narrative description can be used when analyzing scenarios to find system features. You read through the scenarios, look for user actions (usually denoted by active verbs, such as "use," "choose," "send," "update," and so on), and highlight the phrases where these are mentioned. You then think about the product features that can support these actions and how they might be implemented.

In **Table 3.10**, I have done this with Jack's scenario (see **Table 3.5**), in which he sets up a system for his students' project work.

Table 3.10 Jack's scenario with highlighted phrases

Jack is a primary school teacher in Ullapool, teaching P6 pupils. He has decided that a class project should be focused around the fishing industry in the area, looking at the history, development, and economic impact of fishing.

As part of this, students are asked to gather and share reminiscences from relatives, use newspaper archives, and collect old photographs related to fishing and fishing communities in the area. *Students use an iLearn wiki* to gather together fishing stories and *SCRAN (a history archive) to access newspaper archives and photographs.* However, Jack also needs a photo-sharing site as he wants *pupils to take and comment on each others' photos and to upload scans of old photographs* that they may have in their families. He needs to be able to moderate posts with photos before they are shared, because pre-teen children can't understand copyright and privacy issues.

Jack *sends an email to a primary school teachers' group* to see if anyone can recommend an appropriate system. Two teachers reply and both suggest that he use KidsTakePics, a photo-sharing site that allows teachers to check and moderate content. As KidsTakePics *is not integrated with the iLearn authentication service,* he sets up a teacher and a class account with KidsTakePics.

He uses the the iLearn setup service to add KidsTakePics to the services seen by the students in his class so that when they log in, they can immediately use the system to upload photos from their phones and class computers.

The highlighted text identifies features and feature characteristics that should be part of the iLearn system:

- a wiki for group writing;

- access to the SCRAN history archive, which is a shared national resource that provides access to historical newspaper and magazine articles for schools and universities;
- the ability to set up and access an email group;
- the ability to integrate some applications with the iLearn authentication service.

It also confirms the need for the configuration feature that has already been identified from the product vision.

Feature identification should be a team activity, and as features are identified, the team should discuss them and generate ideas about related features. Jack's scenario suggests that there is a need for groups to write together. You should therefore think about age-appropriate ways to design features for:

- collaborative writing, where several people can work simultaneously on the same document;
- blogs and web pages as a way of sharing information.

You can also think about generalizing the features suggested by the scenario. The scenario identifies the need for access to an external archive (SCRAN). However, perhaps the feature that you add to the software should support access to any external archive and allow students to transfer information to the iLearn system.

You can go through a similar process of highlighting phrases in all of the scenarios that you have developed and using them to identify and then generalize a set of product features. If you have developed user stories to refine your scenarios, these may immediately suggest a product feature or feature characteristic. For example, this story was derived from Emma's scenario (see **Table 3.6**):

As a teacher and a parent, I want to be able to select the appropriate iLearn account so that I don't have to have separate credentials for each account.

This story states that the account feature of the iLearn system has to accommodate the idea that a single user may have multiple accounts. Each account is associated with a particular role the user may adopt when using the system. When logging in, users should be able to select the account they wish to use.

3.4.2 The feature list

The output of the feature identification process should be a list of features that you use for designing and implementing your product. There is no need to go into a lot of detail about the features at this stage. You add detail when you are implementing the feature.

You may describe the features on the list using the input/action/output model that I showed in **Figure 3.2**. Alternatively, you can use a standard template, which includes a narrative description of the feature, constraints that have to be considered, and other relevant comments.

Figure 3.11 is an example of this feature template that is used to describe the system authentication feature in the iLearn system.

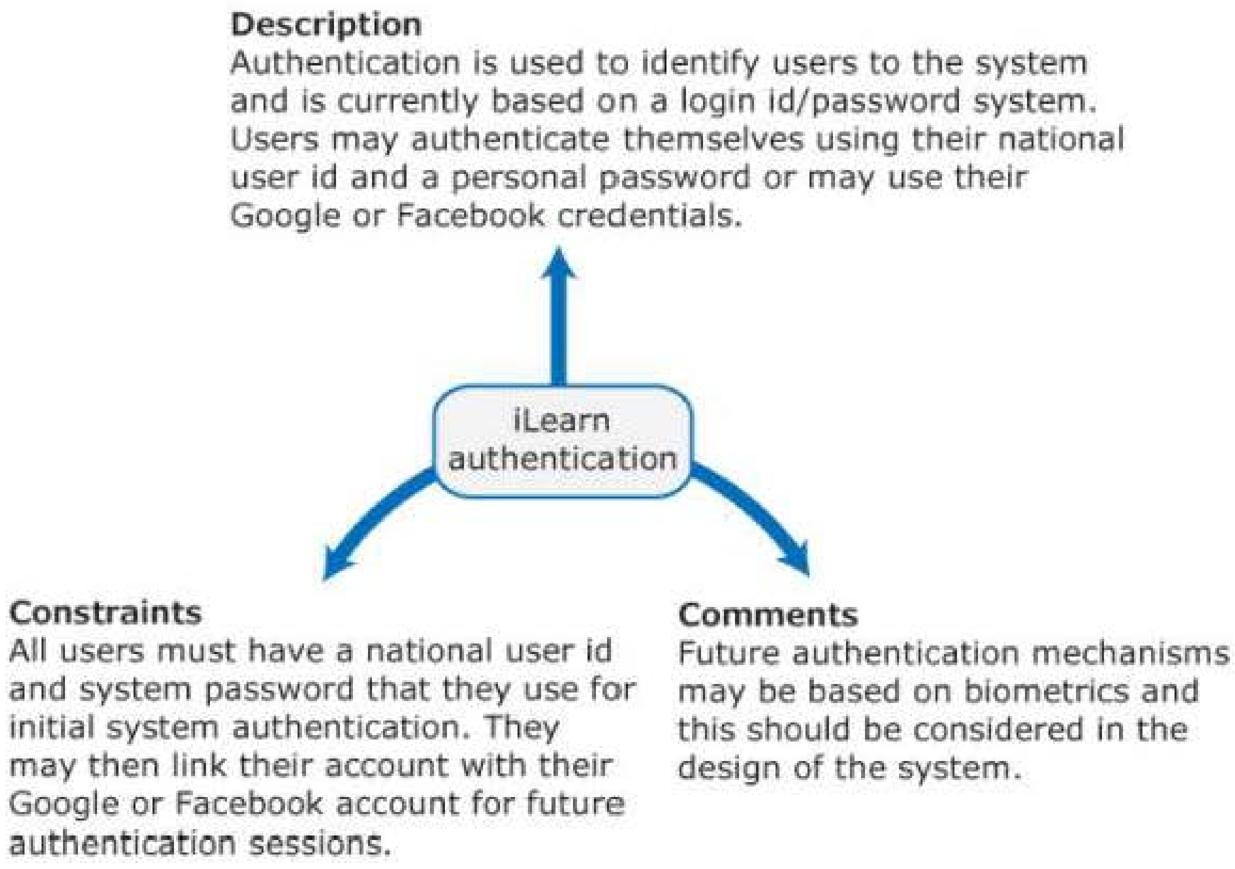

Description
Authentication is used to identify users to the system and is currently based on a login id/password system. Users may authenticate themselves using their national user id and a personal password or may use their Google or Facebook credentials.
iLearn authentication
Constraints
All users must have a national user id and system password that they use for initial system authentication. They may then link their account with their Google or Facebook account for future authentication sessions.
Comments
Future authentication mechanisms may be based on biometrics and this should be considered in the design of the system.

Figure 3.11
The iLearn authentication feature

The descriptions associated with the feature can sometimes be very simple. For example, a Print feature might be described using the simple feature template that I introduced at the beginning of the chapter:

*Print **the** document **to** a selected printer **or to** PDF.*

Alternatively, you can describe a feature from one or more user stories. Descriptions based on user stories are particularly useful if you intend to use Scrum and story-based planning when developing the software.

Table 3.11 shows how you can describe the configuration feature of the iLearn system using user stories and the feature template shown in **Figure 3.11**. In this example, I have used an alternative text-based form of the feature template. This is useful when you have relatively long feature descriptions. Notice that the table includes user stories from the system manager and a teacher.

Table 3.11 Feature description using user stories

iLearn system configuration
Description
As a system manager, I want to create and configure an iLearn environment by adding and removing services to/from that environment so that I can create environments for specific purposes.
As a system manager, I want to set up sub-environments that include a subset of services that are included in another environment.
As a system manager, I want to assign administrators to created environments.
As a system manager, I want to limit the rights of environment administrators so that they cannot accidentally or deliberately disrupt the operation of key services.
As a teacher, I want to be able to add services that are not integrated with the iLearn authentication system.
Constraints
The use of some tools may be limited for license reasons so there may be a need to access license management tools during configuration.
Comments
Based on Elena's and Jack's scenarios

The product development team should meet to discuss the scenarios and stories, and it makes sense to set out the initial list of features on a whiteboard. This can be done using a web-based discussion, but these are less effective than a face-to-face meeting. The feature list should then be recorded in a shared document such as a wiki, a Google Sheet, or an issue-tracking system such as JIRA. Feature descriptions may then be updated and shared as new information about the features emerges.

When you have developed an initial list of feature ideas, you should either extend your existing prototype or create a prototype system to demonstrate these features. As I said in **Chapter 1**, the aim of software prototyping is to test and clarify product ideas and to demonstrate your product to management, funders, and potential customers. You should focus on the novel and critical features of your system. You don't need to implement or demonstrate routine features such as Cut and Paste or Print.

When you have a prototype and have experimented with it, you will inevitably discover problems, omissions, and inconsistencies in your initial list of features. You then update and change this list before moving on to the development of your software product.

I think that scenarios and user stories should always be your starting point for identifying product features. However, the problem with basing product designs on user modeling and research is that it locks in existing ways of working. Scenarios tell you how users work at the moment; they don't show how they might change their ways of working if they had the right software to support them.

User research, on its own, rarely helps you innovate and invent new ways of working. Famously, Nokia, then the world leader in mobile (cell) phones, did extensive user research and produced better and better conventional phones. Then Apple invented the smartphone without user research, and Nokia is now a minor player in the phone business.

As I said, stories and scenarios are tools for thinking; the most important benefit of using them is that you gain an understanding of how your software might be used. It makes sense to start by identifying a feature set from stories and scenarios. However, you should also think creatively about alternative or additional features that help users to work more efficiently or to do things differently.

Key Points

- A software product feature is a fragment of functionality that implements something a user may need or want when using the product.
- The first stage of product development is to identify the list of product features in which you name each feature and give a brief description of its functionality.
- Personas are "imagined users"— character portraits of types of users you think might use your product.
- A persona description should paint a picture of a typical product user. It should describe the user's educational background, technology experience, and why they might want to use your product.
- A scenario is a narrative that describes a situation where a user is accessing product features to do something that they want to do.
- Scenarios should always be written from the user's perspective and should be based on identified personas or real users.
- User stories are finer-grain narratives that set out, in a structured way, something that a user wants from a software system.
- User stories may be used to extend and add detail to a scenario or as part of the description of system features.
- The key influences in feature identification and design are user research, domain knowledge, product knowledge, and technology knowledge.
- You can identify features from scenarios and stories by highlighting user actions in these narratives and thinking about the features that you need to support these actions.

Recommended Reading

"An Introduction to Feature-Driven Development" This article is an introduction to this agile method that focuses on features, a key element of software products. (S. Palmer, 2009)

https://dzone.com/articles/introduction-feature-driven

"A Closer Look at Personas: What they are and how they work" This excellent article on personas explains how they can be used in different situations. Lots of links to relevant associated articles. (S. Golz, 2014)

https://www.smashingmagazine.com/2014/08/a-closer-look-at-personas-part-1/

"How User Scenarios Help to Improve Your UX" Scenarios are often used in designing the user experience for a system. However, the advice here is just as relevant for scenarios intended to help discover system features. (S. Idler, 2011)

https://usabilla.com/blog/how-user-scenarios-help-to-improve-your-ux/

"10 Tips for Writing Good User Stories" Sound advice on story writing is presented by an author who takes a pragmatic view of the value of user stories. (R. Pichler, 2016)

http://www.romanpichler.com/blog/10-tips-writing-good-user-stories/

"What Is a Feature? A qualitative study of features in industrial software product lines" This academic paper discusses a study of features in four different systems and tries to establish what a “good” feature is. It concludes that good features should describe customer-related functionality precisely. (T. Berger et al., 2015)

https://people.csail.mit.edu/mjulia/publications/What_Is_A_Feature_2015.pdf

Presentations, Videos, and Links

https://iansommerville.com/engineering-software-products/features-scenarios-and-stories

Exercises

3.1. Using the input/action/output template that I introduced at the beginning of this chapter, describe two features of software that you commonly use, such as an editor or a presentation system.
3.2. Explain why it is helpful to develop a number of personas representing types of system user before you move on to write scenarios of how the system will be used.
3.3. Based on your own experience of school and teachers, write a persona for a high school science teacher who is interested in building simple electronic systems and using them in class teaching.
3.4. Extend Jack's scenario, shown in **Table 3.5**, to include a section in which students record audio reminiscences from their older friends and relatives and include them in the iLearn system.
3.5. What do you think are the weaknesses of scenarios as a way of envisaging how users might interact with a software system?
3.6. Starting with Jack's scenario (**Table 3.5**), derive four user stories about the use of the iLearn system by both students and teachers.
3.7. What do you think are the weaknesses of user stories when used to identify system features and how they work?
3.8. Explain why domain knowledge is important when identifying and designing product features.
3.9. Suggest how a development team might avoid feature creep when "it is" to be in agreement with "a team" faced with many different suggestions for new features to be added to a product.
3.10. Based on Elena's scenario, shown in **Table 3.7**, use the method of highlighting phrases in the scenario to identify four features that might be included in the iLearn system.

4 Software Architecture

The focus of this book is software products—individual applications that run on servers, personal computers, or mobile devices. To create a reliable, secure, and efficient product, you need to pay attention to its overall organization, how the software is decomposed into components, the server organization, and the technologies used to build the software. In short, you need to design the software architecture.

The architecture of a software product affects its performance, usability, security, reliability, and maintainability. Architectural issues are so important that three chapters in this book are devoted to them. In this chapter I discuss the decomposition of software into components, client–server architecture, and technology issues that affect the software architecture. In **Chapter 5** I cover architectural issues and choices you have to make when implementing your software on the cloud. In **Chapter 6** I cover microservices architecture, which is particularly important for cloud-based products.

If you google "software architecture definition," you find many different interpretations of the term. Some focus on "architecture" as a noun, the structure of a system; others consider "architecture" as a verb, the process of defining these structures. Rather than try to invent yet another definition, I use a definition of software architecture that is included in an IEEE standard,[1] shown in **Table 4.1**.

1IEEE standard 1471. This has now been superseded by a later standard that has revised the definition. In my opinion, the revised definition is not an improvement and it is harder to explain and understand. **https://en.wikipedia.org/wiki/IEEE_1471**

Table 4.1 The IEEE definition of software architecture

Software architecture

Architecture is the fundamental organization of a software system embodied in its components, their relationships to each other and to the environment, a nd the principles guiding its design and evolution.

An important term in this definition is "components." Here it is used in a very general way, so a component can be anything from a program (large scale) to an object (small scale). A component is an element that implements a coherent set of functionality or features. When designing software architecture, you don't have to decide how an architectural element or component is to be implemented. Rather, you design the component interface and leave the implementation of that interface to a later stage of the development process.

The best way to think of a software component is as a collection of one or more services that may be used by other components (**Figure 4.1**). A service is a coherent fragment of functionality. It can range from a large-scale service, such as a database service, to a microservice, which does one very specific thing. For example, a microservice might simply check the validity of a URL. Services can be implemented directly, as I discuss in **Chapter 6**, which covers microservice architecture. Alternatively, services can be part of modules or objects and accessed through a defined component interface or an application programming interface (API).

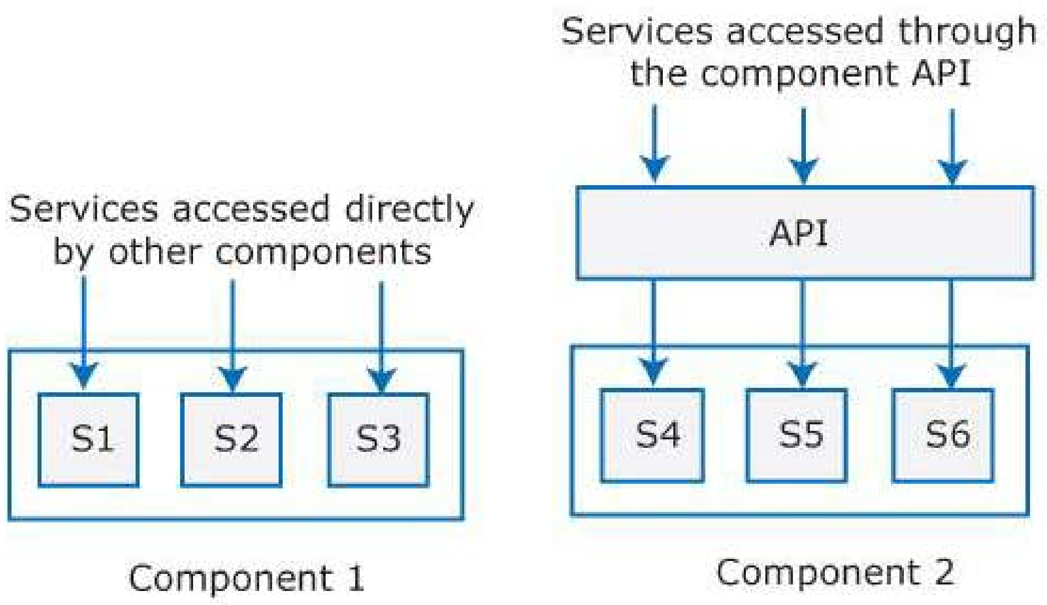


Figure 4.1
Access to services provided by software components

The initial enthusiasts for agile development invented slogans such as "You Ain't Gonna Need It" (YAGNI) and "Big Design Up Front" (BDUF), where YAGNI is good and BDUF is bad. They suggested that developers should not plan for change in their systems because change can't be predicted and might never happen. Many people think this means that agile developers believed there was no need to design the architecture of a software system before implementation. Rather, when issues emerged during development, they should simply be tackled by refactoring—changing and reorganizing the software.

The inventors of agile methods are good engineers, and I don't think they intended that software architecture should not be designed. A principle of agile methods is that system planning and documentation should be minimized. However, this does not mean that you can't plan and describe the overall structure of your system. Agile methods now recognize the importance of architectural design and suggest that this should be an early activity in the development process. You can do this in a Scrum sprint where the outcome of the sprint is an informal architectural design.

Some people think it is best to have a single software architect. This person should be an experienced engineer who uses background knowledge and expertise to create a coherent architecture. However, the problems with this “single architect” model is that the team may not understand the architectural decisions that were made. To make sure that your whole team understands the architecture, I think everyone should be involved, in some way, in the architectural design process. This helps less experienced team members learn and understand why decisions are made. Furthermore, new team members may have knowledge and insights into new or unfamiliar technologies that can be used in the design and implementation of the software.

A development team should design and discuss the software product architecture before starting the final product implementation. They should agree on priorities and understand the trade-offs that they are making in these architectural decisions. They should create a description of the product architecture that sets out the fundamental structure of the software and serves as a reference for its implementation.

4.1 Why is architecture important?

I suggested in Chapter 1 that you should always develop a product prototype. The aim of a prototype is to help you understand more about the product that you are developing, and so you should aim to develop this as quickly as possible. Issues such as security, usability, and long-term maintainability are not important at this stage.

When you are developing a final product, however, "non-functional" attributes are critically important (Table 4.2). It is these attributes, rather than product features, that influence user judgements about the quality of your software. If your product is unreliable, insecure, or difficult to use, then it is almost bound to be a failure. Product development takes much longer than prototyping because of the time and effort that are needed to ensure that your product is reliable, maintainable, secure, and so on.

Table 4.2 Non-functional system quality attributes

Attribute	Key issue
Responsiveness	Does the system return results to users in a reasonable time?
Reliability	Do the system features behave as expected by both developers and users?
Availability	Can the system deliver its services when requested by users?
Security	Does the system protect itself and users' data from unauthorized attacks and intrusions?

Usability	Can system users access the features that they need and use them quickly and without errors?
Maintainability	Can the system be readily updated and new features added without undue costs?
Resilience	Can the system continue to deliver user services in the event of partial failure or external attack?

Architecture is important because the architecture of a system has a fundamental influence on these non-functional properties. **Table 4.3** is a non-computing example of how architectural choices affect system properties. It is taken from the film *Rogue One*, part of the *Star Wars* saga.

Table 4.3 The influence of architecture on system security

A centralized security architecture
In the *Star Wars* prequel *Rogue One* (**https://en.wikipedia.org/wiki/Rogue_One**), the evil Empire has stored the plans for all of their equipment in a single, highly secure, well-guarded, remote location. This is called a centralized security architecture. It is based on the principle that if you maintain all of your information in one place, then you can apply lots of resources to protect that information and ensure that intruders can't get it. Unfortunately (for the Empire), the rebels managed to breach their security. They stole the plans for the Death Star, an event that underpins the whole *Star Wars* saga. In trying to stop them, the Empire destroyed their entire archive of system documentation with who knows what resultant costs. Had the Empire chosen a distributed security architecture, with different parts of the Death Star plans stored in different locations, then stealing the plans would have be

en more difficult. The rebels would have had to breach security in all locations to steal the complete Death Star blueprints.

Rogue One is science fiction, but it demonstrates that architectural decisions have fundamental consequences. The benefits of a centralized security architecture are that it is easy to design and build protection and the protected information can be accessed efficiently. However, if your security is breached, you lose everything. If you distribute information, it takes longer to access all of the information and costs more to protect it. If security is breached in one location, however, you lose only the information that you have stored there.

Figures 4.2 and **4.3** illustrate a situation where the system architecture affects the maintainability and performance of a system. **Figure 4.2** shows a system with two components (C1 and C2) that share a common database. This is a common architecture for web-based systems. Let's assume that C1 runs slowly because it has to reorganize the information in the database before using it. The only way to make C1 faster might be to change the database. This means that C2 also has to be changed, which may potentially affect its response time.

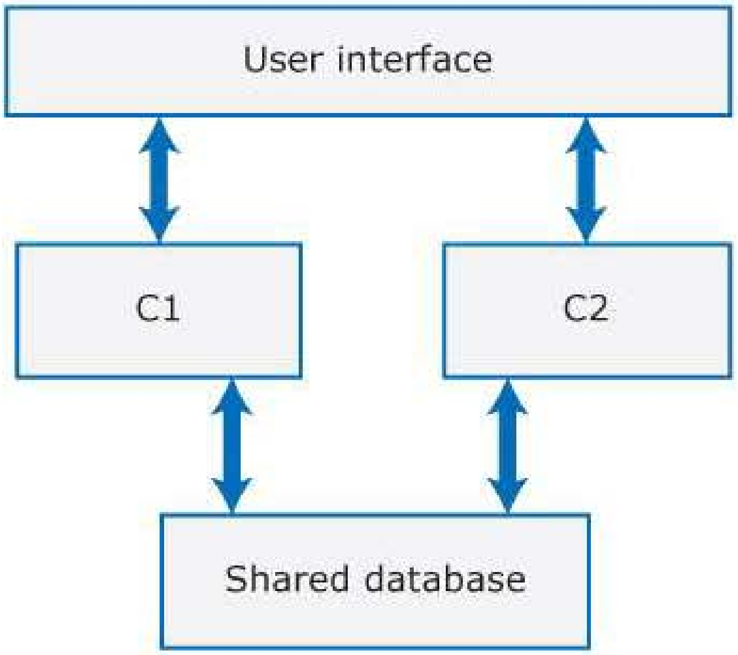


Figure 4.2

Shared database architecture

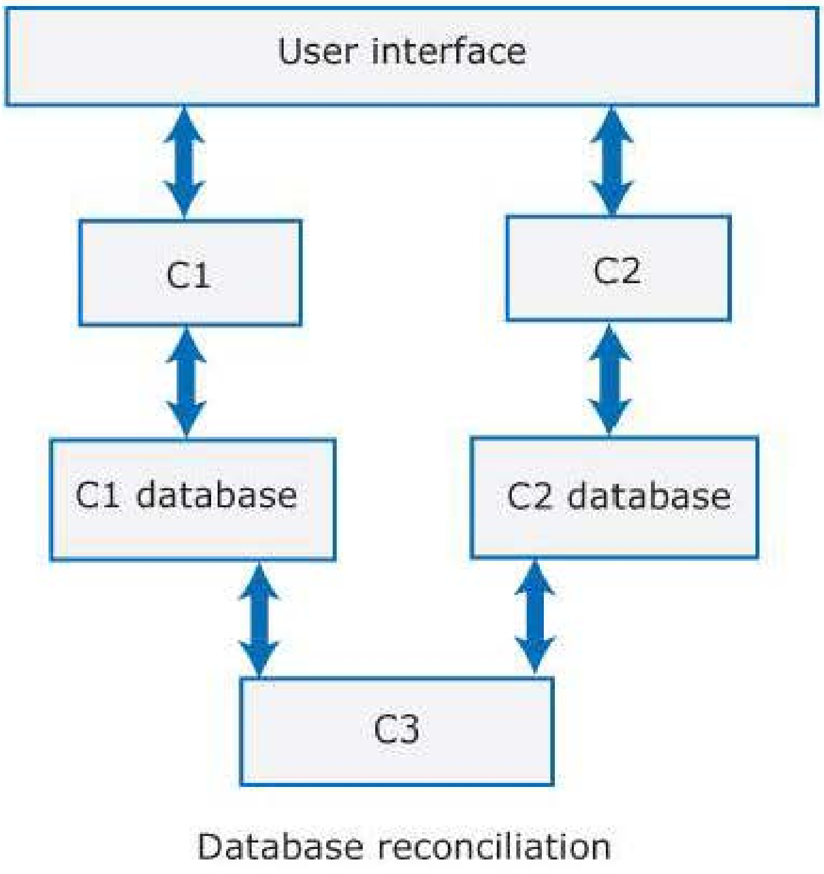


Figure 4.3
Multiple database architecture

Figure 4.3 shows a different architecture where each component has its own copy of the parts of the database that it needs. Each of these components can therefore use a different database structure, and so operate efficiently. If one component needs to change the database organization, this does not affect the other component. Also, the system can continue to provide a partial service in the event of a database failure. This is impossible in a centralized database architecture.

However, the distributed database architecture may run more slowly and may cost more to implement and change. There needs to be a mechanism (shown here as component C3) to ensure that the data shared by C1 and C2 are kept consistent when changes are made. This takes time and it is possible that users will occasionally see inconsistent information. Furthermore, additional storage costs are associated with the distributed database architecture and higher costs of change if a new component that requires its own database has to be added to the system.

It is impossible to optimize all of the non-functional attributes in the same system. Optimizing one attribute, such as security, inevitably affects other attributes, such as system usability and efficiency. You have to think about these issues and the software architecture before you start programming. Otherwise, it is almost inevitable that your product will have undesirable characteristics and will be difficult to change.

Another reason why architecture is important is that the software architecture you choose affects the complexity of your product. The more complex a system, the more difficult and expensive it is to understand and change. Programmers are more likely to make mistakes and introduce bugs and security vulnerabilities when they are modifying or extending a complex system. Therefore, minimizing complexity should be an important goal for architectural design.

The organization of a system has a profound effect on its complexity, and it is very important to pay attention to this when designing the software architecture. I explain architectural complexity in **Section 4.3**, and I cover general issues of program complexity in **Chapter 8**.

4.2 Architectural design

Architectural design involves understanding the issues that affect the architecture of your particular product and creating a description of the architecture that shows the critical components and some of their relationships. The architectural issues that are most important for software product development are shown in **Figure 4.4** and **Table 4.4**.

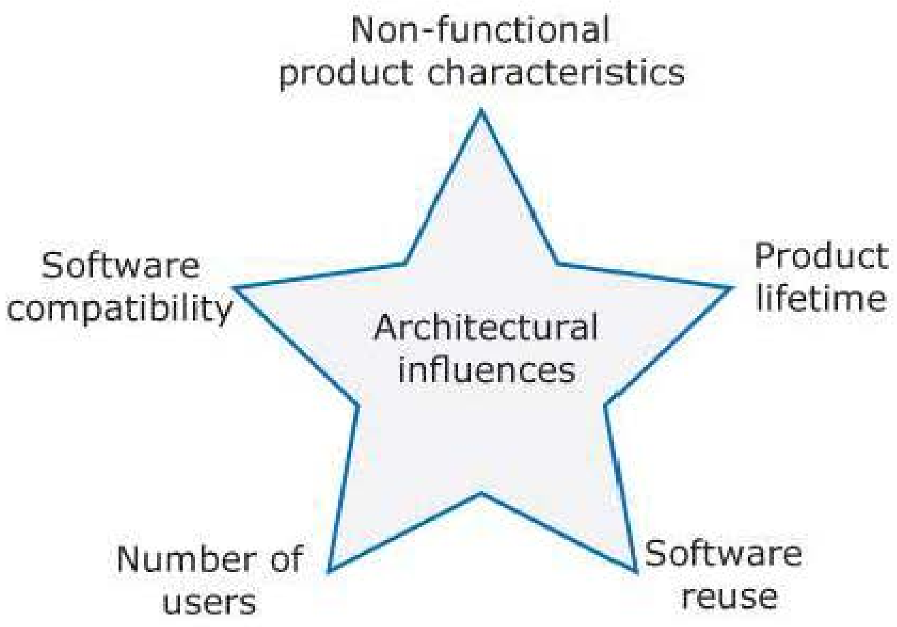


Figure 4.4
Issues that influence architectural decisions

Table 4.4 The importance of architectural design issues

Issue	Architectural importance
Non-functional product characterist ics	Non-functional product characteristi cs such as security and performance affect all users. If you get these wron g, your product is unlikely to be a co mmercial success. Unfortunately, so

	me characteristics are opposing, so you can optimize only the most important.
Product lifetime	If you anticipate a long product lifetime, you need to create regular product revisions. You therefore need an architecture that can evolve, so that it can be adapted to accommodate new features and technology.
Software reuse	You can save a lot of time and effort if you can reuse large components from other products or open-source software. However, this constrains your architectural choices because you must fit your design around the software that is being reused.
Number of users	If you are developing consumer software delivered over the Internet, the number of users can change very quickly. This can lead to serious performance degradation unless you design your architecture so that your system can be quickly scaled up and down.
Software compatibility	For some products, it is important to maintain compatibility with other software so that users can adopt your product and use data prepared using a different system. This may limit architectural choices, such as the database software that you can use.

Other human and organizational factors also affect architectural design decisions. These include the planned schedule to bring the product to market, the capabilities of your development team, and the software development budget. If you choose an architecture that requires your team to learn unfamiliar technologies, then this may delay the delivery of your system. There is no point in creating a "perfect" architecture that is delivered late if this means that a competing product captures the market.

Architectural design involves considering these issues and deciding on essential compromises that allow you to create a system that is "good enough" and can be delivered on time and on budget. Because it is impossible to optimize everything, you have to make a series of trade-offs when choosing an architecture for your system. Some examples are:

- maintainability vs. performance;
- security vs. usability;
- availability vs. time to market and cost.

System maintainability is an attribute that reflects how difficult and expensive it is to make changes to a system after it has been released to customers. In general, you improve maintainability by building a system from small self-contained parts, each of which can be replaced or enhanced if changes are required. Wherever possible, you should avoid shared data structures and you should make sure that, when data are processed, separate components are used to "produce" and to "consume" data.

In architectural terms, this means that the system should be decomposed into fine-grain components, each of which does one thing and one thing only. More general functionality emerges by creating networks of these components that communicate and exchange information. Microservice architectures, explained in Chapter 6, are an example of this type of architecture.

However, it takes time for components to communicate with each other. Consequently, if many components are involved in implementing a product feature, the software will be slower. Avoiding shared data structures also has an impact on performance. There may be delays involved in transferring data from one component to another and in ensuring that duplicated data are kept consistent.

The constant and increasing risk of cybercrime means that all product developers have to design security into their software. Security is so important for product development that I devote a separate chapter (**Chapter 7**) to this topic. You can achieve security by designing the system protection as a series of layers (**Figure 4.5**). An attacker has to penetrate all of those layers before the system is compromised. Layers might include system authentication layers, a separate critical feature authentication layer, an encryption layer, and so on. Architecturally, you can implement each of these layers as separate components so that if an attacker compromises one of these components, then the other layers remain intact.

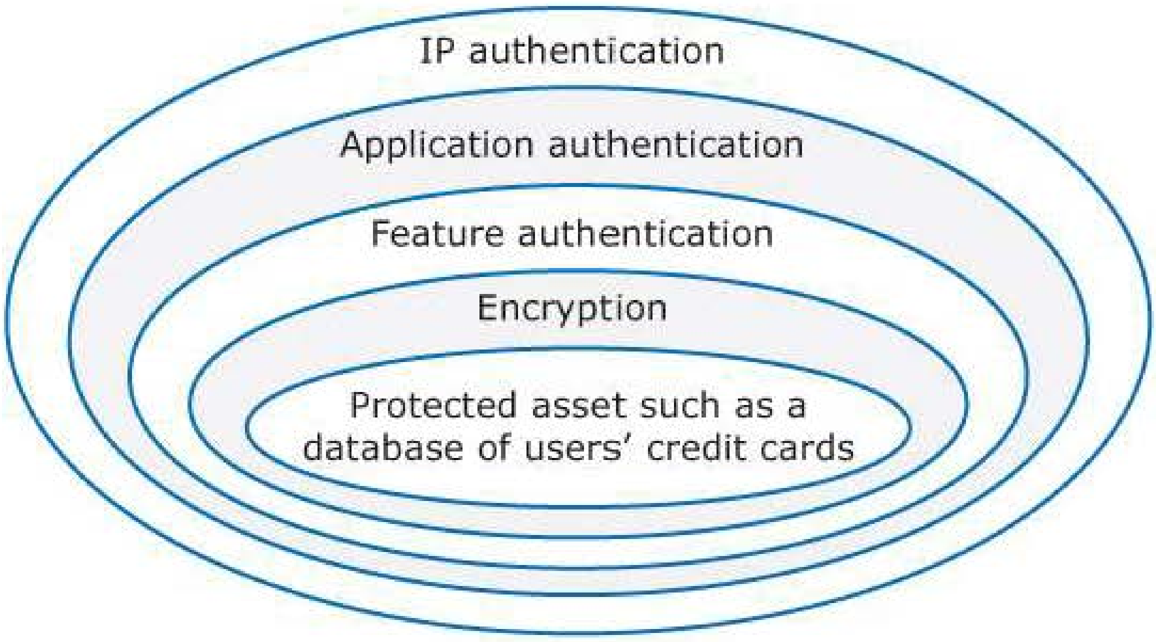

IP authentication
Application authentication
Feature authentication
Encryption
Protected asset such as a database of users' credit cards

Figure 4.5
Authentication layers

Unfortunately, there are drawbacks to using multiple authentication layers. A layered approach to security affects the usability of the software. Users have to remember information, like passwords, that is needed to penetrate a security layer. Their interaction with the system is inevitably slowed by its security features. Many users find this irritating and often look for work-arounds so that they do not have to re-authenticate to access system features or data.

Many security breaches arise because users behave in an insecure way, such as choosing passwords that are easy to guess, sharing passwords, and leaving systems logged on. They do this because they are frustrated by system security features that are difficult to use or that slow down their access to the system and its data. To avoid this, you need an architecture that doesn't have too many security layers, that doesn't enforce unnecessary security, and that provides, where possible, helper components that reduce the load on users.

The availability of a system is a measure of the amount of uptime of that system. It is normally expressed as a percentage of the time that a system is available to deliver user services. Therefore, an availability of 99.9% in a system that is intended to be constantly available means that the system should be available for 86,313 seconds out of the 86,400 seconds in a day. Availability is particularly important in enterprise products, such as products for the finance industry, where 24/7 operation is expected.

Architecturally, you improve availability by having redundant components in a system. To make use of redundancy, you include sensor components that detect failure and switching components that switch operation to a redundant component when a failure is detected. The problem here is that implementing these extra components takes time and increases the cost of system development. It adds complexity to the system and therefore increases the chances of introducing bugs and vulnerabilities. For this reason, most product software does not use component-switching in the event of system failure. As I explain in

Chapter 8, you can use reliable programming techniques to reduce the changes of system failure.

Once you have decided on the most important quality attributes for your software, you have to consider three questions about the architectural design of your product:

1. How should the system be organized as a set of architectural components, where each of these components provides a subset of the overall system functionality? The organization should deliver the system security, reliability, and performance that you need.
2. How should these architectural components be distributed and communicate with each other?
3. What technologies should be used in building the system, and what components should be reused?

I cover these three questions in the remaining sections of this chapter.

Architectural descriptions in product development provide a basis for the development team to discuss the organization of the system. An important secondary role is to document a shared understanding of what needs to be developed and what assumptions have been made in designing the software. The final system may differ from the original architectural model, so it is not a reliable way of documenting delivered software.

I think informal diagrams based around icons to represent entities, lines to represent relationships, and text are the best way to describe and share information about software product architectures. Everyone can participate in the design process. You can draw and change informal diagrams quickly without using special software tools. Informal notations are flexible so that you can make unanticipated changes easily. New people joining a team can understand them without specialist knowledge.

The main problems with informal models are that they are ambiguous and they can’t be checked automatically for omissions and inconsistencies. If you use more formal approaches, based on architectural description languages (ADLs) or the Unified Modeling Language (UML), you reduce ambiguity and can use checking tools. However, my experience is that formal notations get in the way of the creative design process. They constrain expressiveness and require everyone to understand them before they can participate in the design process.

4.3 System decomposition

The idea of abstraction is fundamental to all software design. Abstraction in software design means that you focus on the essential elements of a system or software component without concern for its details. At the architectural level, your concern should be on large-scale architectural components. Decomposition involves analyzing these large-scale components and representing them as a set of finer-grain components.

For example, **Figure 4.6** is a diagram of the architecture of a product that I was involved with some years ago. This system was designed for use in libraries and gave users access to documents that were stored in a number of private databases, such as legal and patent databases. Payment was required for access to these documents. The system had to manage the rights to these documents and collect and account for access payments.

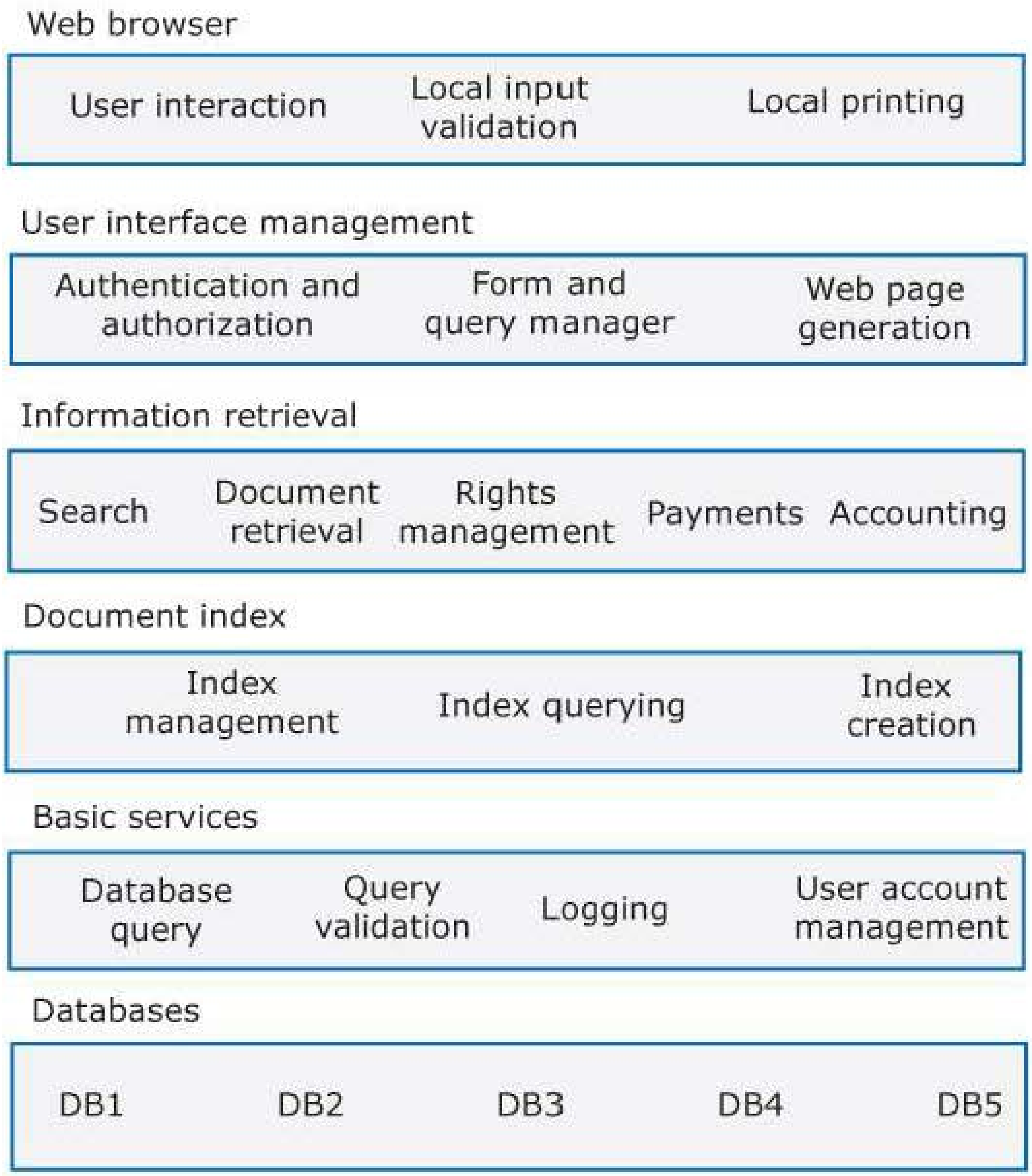


Figure 4.6
An architectural model of a document retrieval system

In this diagram, each layer in the system includes a number of logically related components. Informal layered models, like **Figure 4.6**, are widely used to show how a system is decomposed into components, with each component providing significant system functionality.

Web-based and mobile systems are event-based systems. An event in the user interface, such as a mouse click, triggers the actions to implement the user's choice. This means that the flow of control in a layered system is top-down. User events in the higher layers trigger actions in that layer that, in turn, trigger events in lower layers. By contrast, most information flows in the system are bottom-up. Information is created at lower layers, is transformed in the intermediate layers, and is finally delivered to users at the top level.

There is often confusion about architectural terminology, words such as "service," "component," and "module." There are no standard, widely accepted definitions of these terms, but I try to use them consistently in this chapter and elsewhere in the book:

1. A *service* is a coherent unit of functionality. This may mean different things at different levels of the system. For example, a system may offer an email service and this email service may itself include services for creating, sending, reading, and storing email.
2. A *component* is a named software unit that offers one or more services to other software components or to the end-users of the software. When used by other components, these services are accessed through an API. Components may use several other components to implement their services.
3. A *module* is a named set of components. The components in a module should have something in common. For example, they may provide a set of related services.

Complexity in a system architecture arises because of the number and the nature of the relationships among components in that system. I discuss this in more detail in **Chapter 8**. When you change a program, you have to understand these relationships to know how changes to one component affect other components. When decomposing a system into components, you should try to avoid introducing unnecessary complexity into the software.

Components have different types of relationships with other components (**Figure 4.7**). Because of these relationships, when you make a change to one component, you often need to make changes to several other components.

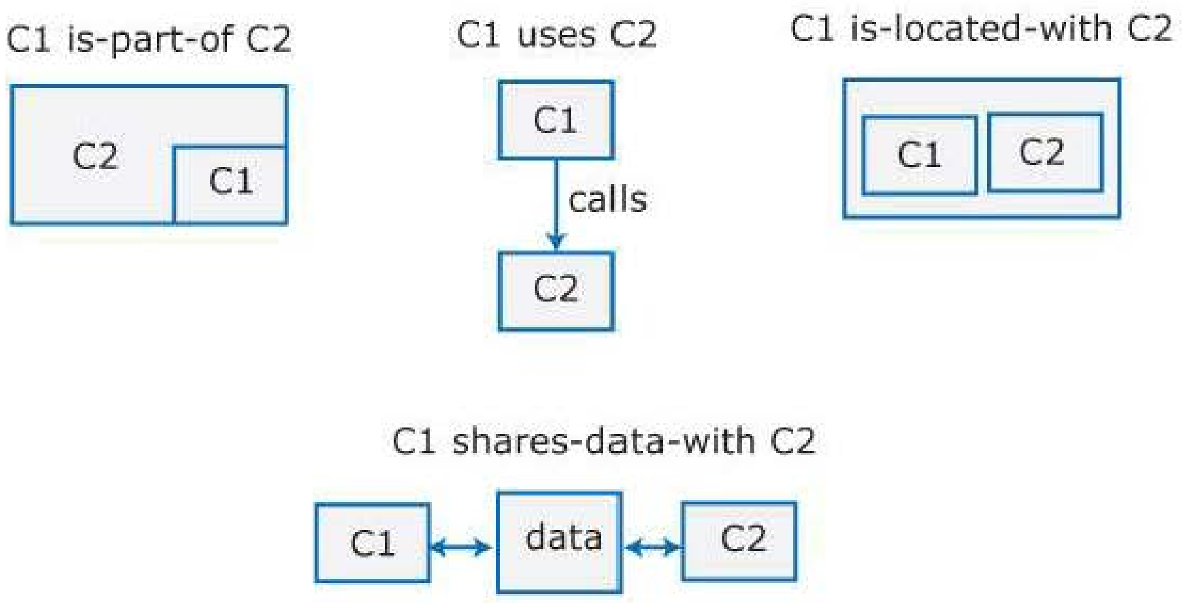


Figure 4.7
Examples of component relationships

Figure 4.7 shows four types of component relationship:

1. **Part-of** One component is part of another component. For example, a function or method may be part of an object.
2. **Uses** One component uses the functionality provided by another component.
3. **Is-located-with** One component is defined in the same module or object as another component.
4. **Shares-data-with** A component shares data with another component.

As the number of components increases, the number of relationships tends to increase at a faster rate. This is the reason large systems are more complex than small systems. It is impossible to avoid complexity increasing with the size of the software. However, you can control architectural complexity by doing two things:

1. **Localize relationships** If there are relationships between components A and B (say), they are easier to understand if A and B are defined in the same module. You should identify logical component groupings (such as the layers in a layered architecture) with relationships mostly within a component group.
2. **Reduce shared dependencies** Where components A and B depend on some other component or data, complexity increases because changes to the shared component mean you have to understand how these changes affect both A and B. It is always preferable to use local data wherever possible and to avoid sharing data if you can.

Three general design guidelines help to control complexity, as shown in Figure 4.8.

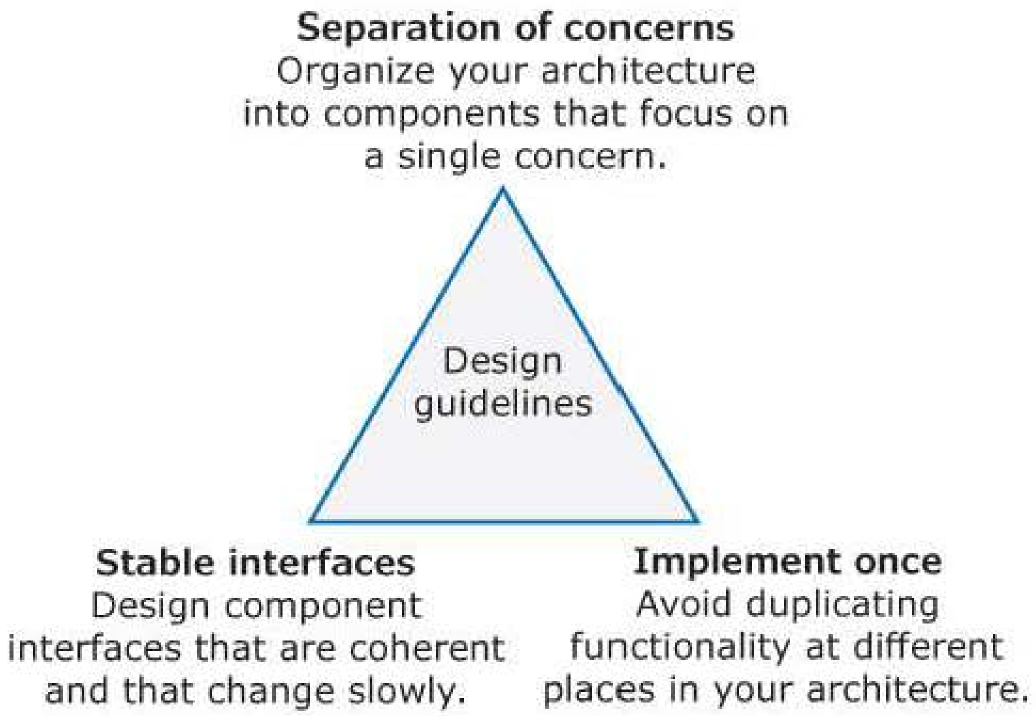

Separation of concerns
Organize your architecture into components that focus on a single concern.
Design guidelines
Stable interfaces
Design component interfaces that are coherent and that change slowly.
Implement once
Avoid duplicating functionality at different places in your architecture.

Figure 4.8
Architectural design guidelines

The *separation of concerns* guideline suggests that you should identify relevant architectural concerns in groupings of related functionality.

Examples of architectural concerns are user interaction, authentication, system monitoring, and database management. Ideally, you should be able to identify the components or groupings of components in your architecture that are related to each concern. At a lower level, separation of concerns means that components should, ideally, do only one thing. I cover separation of concerns in more detail in **Chapter 8**.

The *implement once* guideline suggests that you should not duplicate functionality in your software architecture. This is important, as duplication can cause problems when changes are made. If you find that more than one architectural component needs or provides the same or a similar service, you should reorganize your architecture to avoid duplication.

You should never design and implement software where components know of and rely on the implementation of other components. Implementation dependencies mean that if a component is changed, then the components that rely on its implementation also have to be changed. Implementation details should be hidden behind a component interface (API).

The *stable interfaces* guideline is important so that components that use an interface do not have to be changed because the interface has changed.

Layered architectures, such as the document retrieval system architecture shown in **Figure 4.6**, are based on these general design guidelines:

1. Each layer is an area of concern and is considered separately from other layers. The top layer is concerned with user interaction, the next layer down with user interface management, the third layer with information retrieval, and so on.
2. Within each layer, the components are independent and do not overlap in functionality. The lower layers include components that

provide general functionality, so there is no need to replicate this in the components in a higher level.

3. The architectural model is a high-level model that does not include implementation information. Ideally, components at level X (say) should only interact with the APIs of the components in level X-1; that is, interactions should be between layers and not across layers. In practice, however, this is often impossible without code duplication. The lower levels of the stack of layers provide basic services that may be required by components that are not in the immediate level above them. It makes no sense to add additional components in a higher layer if these are used only to access lower-level components.

Layered models are informal and are easy to draw and understand. They can be drawn on a whiteboard so that the whole team can see how the system is decomposed. In a layered model, components in lower layers should never depend on higher-level components. All dependencies should be on lower-level components. This means that if you change a component at level X in the stack, you should not have to make changes to components at lower levels in the stack. You only have to consider the effects of the change on components at higher levels.

The layers in the architectural model are not components or modules but are simply logical groupings of components. They are relevant when you are designing the system, but you can't normally identify these layers in the system implementation.

The general idea of controlling complexity by localizing concerns within a single layer of an architecture is a compelling one. If you can do this, you don't have to change components in other layers when components in any one layer are modified. Unfortunately, there are two reasons why localizing concerns may not always be possible:

1. For practical reasons of usability and efficiency, it may be necessary to divide functionality so that it is implemented in different layers.

2. Some concerns are cross-cutting concerns and have to be considered at every layer in the stack.

You can see an example of the problem of practical separation of concerns in **Figure 4.6**. The top layer includes "Local input validation" and the fifth layer in the stack includes "Query validation." The "validation concern" is not implemented in a single lower-level server component because this is likely to generate too much network traffic.

If user data validation is a server rather than a browser operation, this requires a network transaction for every field in a form. Obviously, this slows the system down. Therefore, it makes sense to implement some local input checking, such as date checking, in the user's browser or mobile app. Some checking, however, may require knowledge of database structure or a user's permissions, and this can be carried out only when all of the form has been completed. As I explain in **Chapter 7**, the checking of security-critical fields should also be a server-side operation.

Cross-cutting concerns are systemic concerns; that is, they affect the whole system. In a layered architecture, cross-cutting concerns affect all layers in the system as well as the way in which people use the system. **Figure 4.9** shows three cross-cutting concerns—security, performance and reliability—that are important for software products.

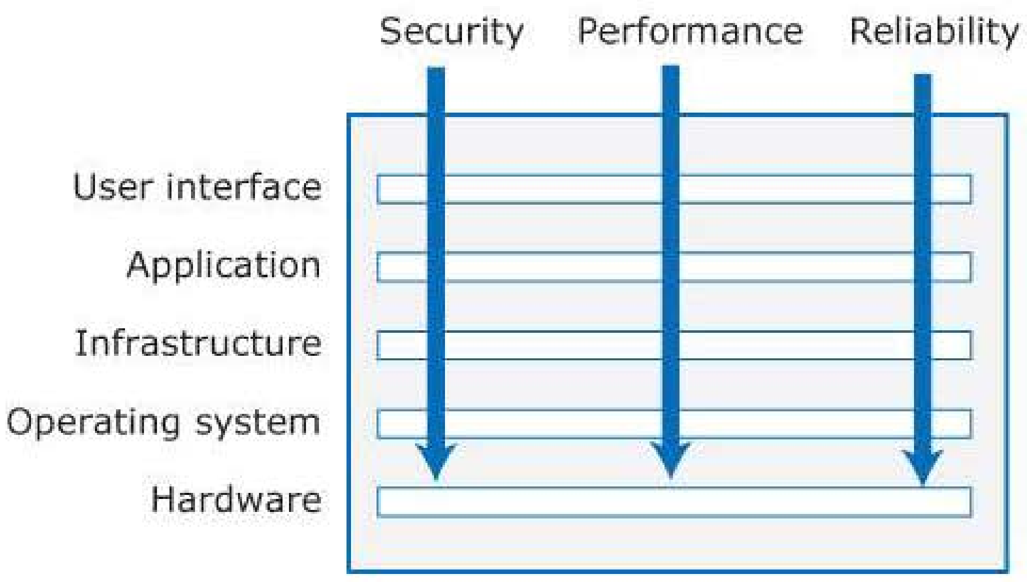


Figure 4.9
Cross-cutting concerns

Cross-cutting concerns are completely different from the functional concerns represented by layers in a software architecture. Every layer has to take them into account, and there are inevitably interactions between the layers because of these concerns. These cross-cutting concerns make it difficult to improve system security after it has been designed. **Table 4.5** explains why security cannot be localized in a single component or layer.

Table 4.5 Security as a cross-cutting concern

Security architecture
Different technologies are used in different layers, such as an SQL database or a Firefox browser. Attackers can try to use vulnerabilities in these technologies to gain access. Consequently, you need protection from attacks at each layer as well as protection at lower layers in the system from successful attacks that have occurred at higher-level layers.
If there is only a single security component in a system, this represents a critical system vulnerability. If all security checking goes through that component and it stops working properly or is compromised in an attack, then you have no reliable security in your system. By distributing security across the layers, y

our system is more resilient to attacks and software failure (remember the *Ro gue One* example earlier in the chapter).

Let's assume that you are a software architect and you want to organize your system into a series of layers to help control complexity. You are then faced with the general question "Where do I start?". Fortunately, many software products that are delivered over the web have a common layered structure that you can use as a starting point for your design. This common structure is shown in **Figure 4.10**. The functionality of the layers in this generic layered architecture is explained in **Table 4.6**.

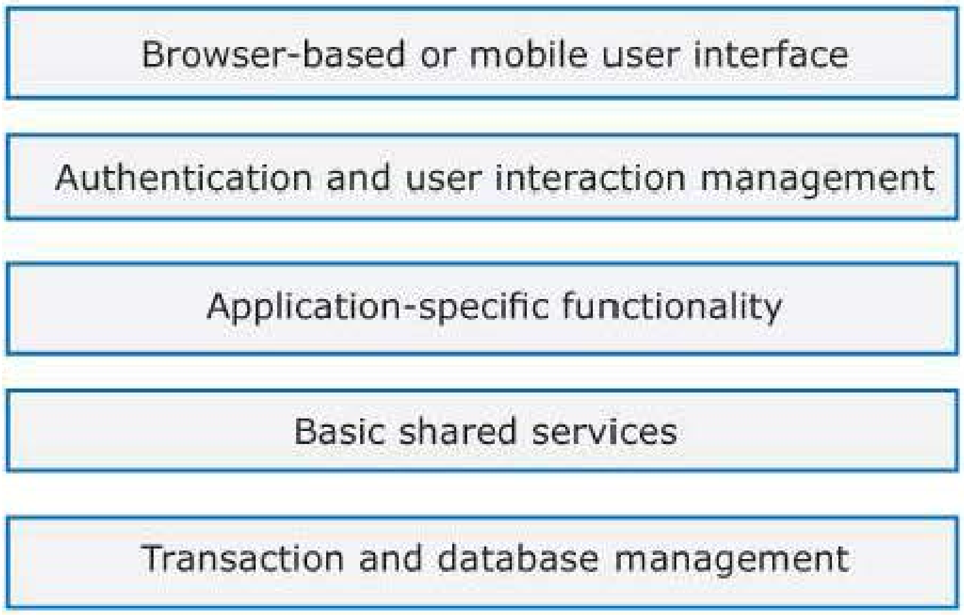

Browser-based or mobile user interface
Authentication and user interaction management
Application-specific functionality
Basic shared services
Transaction and database management

Figure 4.10
A generic layered architecture for a web-based application

Table 4.6 Layer functionality in a web-based application

Layer	Explanation
Browser-based or mobile user interf ace	A web browser system interface in w hich HTML forms are often used to c ollect user input. Javascript compone nts for local actions, such as input va

	lidation, should also be included at t his level. Alternatively, a mobile inter face may be implemented as an app.
Authentication and UI management	A user interface management layer t hat may include components for user authentication and web page generat ion.
Application-specific functionality	An “application” layer that provides functionality of the application. Som etimes this may be expanded into mo re than one layer.
Basic shared services	A shared services layer that includes components that provide services us ed by the application layer componen ts.
Database and transaction manageme nt	A database layer that provides servic es such as transaction management a nd recovery. If your application does not use a database, then this may not be required.

For web-based applications, the layers shown in **Figure 4.10** can be the starting point for your decomposition process. The first stage is to think about whether this five-layer model is the right one or whether you need more or fewer layers. Your aim should be for layers to be logically coherent, so that all components in a layer have something in common. This may mean that you need one or more additional layers for your application-specific functionality. Sometimes you may wish to have authentication in a separate layer, and sometimes it makes sense to integrate shared services with the database management layer.

Once you have figured out how many layers to include in your system, you can start to populate these layers. In my experience, the best way to do this is to involve the whole team and try out various decompositions to help understand their advantages and disadvantages. This is a trial-and-error process; you stop when you have what seems to be a workable decomposition architecture.

The discussion about system decomposition may be driven by fundamental principles that should apply to the design of your application system. These set out goals that you wish to achieve. You can then evaluate architectural design decisions against these goals. For example, **Table 4.7** shows the principles that we thought were most important when designing the iLearn system architecture.

Table 4.7 iLearn architectural design principles

Principle	Explanation
Replaceability	It should be possible for users to repl ace applications in the system with al ternatives and to add new applicatio ns. Consequently, the list of applicati ons included should not be hard-wir ed into the system.
Extensibility	It should be possible for users or syst em administrators to create their ow n versions of the system, which may extend or limit the "standard" syste m.
Age-appropriate	Alternative user interfaces should be supported so that age-appropriate in terfaces for students at different leve ls can be created.
Programmability	It should be easy for users to create t

	heir own applications by linking exis ting applications in the system.
Minimum work	Users who do not wish to change the system should not have to do extra w ork so that other users can make cha nges.

Our goal in designing the iLearn system was to create an adaptable, universal system that could be updated easily as new learning tools became available. This means it must be possible to change and replace components and services in the system (principles 1 and 2). Because the potential system users ranged in age from 3 to 18, we needed to provide age-appropriate user interfaces and make it easy to choose an interface (principle 3). Principle 4 also contributes to system adaptability, and principle 5 was included to ensure that this adaptability did not adversely affect users who did not require it.

Unfortunately, principle 1 may sometimes conflict with principle 4. If you allow users to create new functionality by combining applications, then these combined applications may not work if one or more of the constituent applications are replaced. You often have to address this kind of conflict in architectural design.

These principles led us to an architectural design decision that the iLearn system should be service-oriented. Every component in the system is a service. Any service is potentially replaceable, and new services can be created by combining existing services. Different services that deliver comparable functionality can be provided for students of different ages.

Using services means that the potential conflict I identified above is mostly avoidable. If a new service is created by using an existing service and, subsequently, other users want to introduce an alternative, they may do so. The older service can be retained in the system, so that users of that service don't have to do more work because a newer service has been introduced.

We assumed that only a minority of users would be interested in programming their own system versions. Therefore, we decided to provide a standard set of application services that had some degree of integration with other services. We anticipated that most users would rely on these and would not wish to replace them. Integrated application services, such as blogging and wiki services, could be designed to share information and make use of common shared services. Some users may wish to introduce other services into their environment, so we also allowed for services that were not tightly integrated with other system services.

We decided to support three types of application service integration:

1. **Full integration** Services are aware of and can communicate with other services through their APIs. Services may share system services and one or more databases. An example of a fully integrated service is a specially written authentication service that checks the credentials of system users.
2. **Partial integration** Services may share service components and databases, but they are not aware of and cannot communicate directly with other application services. An example of a partially integrated service is a Wordpress service in which the Wordpress system was changed to use the standard authentication and storage services in the system. Office 365, which can be integrated with local authentication systems, is another example of a partially integrated service that we included in the iLearn system.
3. **Independent** These services do not use any shared system services or databases, and they are unaware of any other services in the system. They can be replaced by any other comparable service. An example of an independent service is a photo management system that maintains its own data.

The layered model for the iLearn system that we designed is shown in **Figure 4.11**. To support application "replaceability", we did not base the system around a shared database. However, we assumed that fully-integrated applications would use shared services such as storage and authentication.

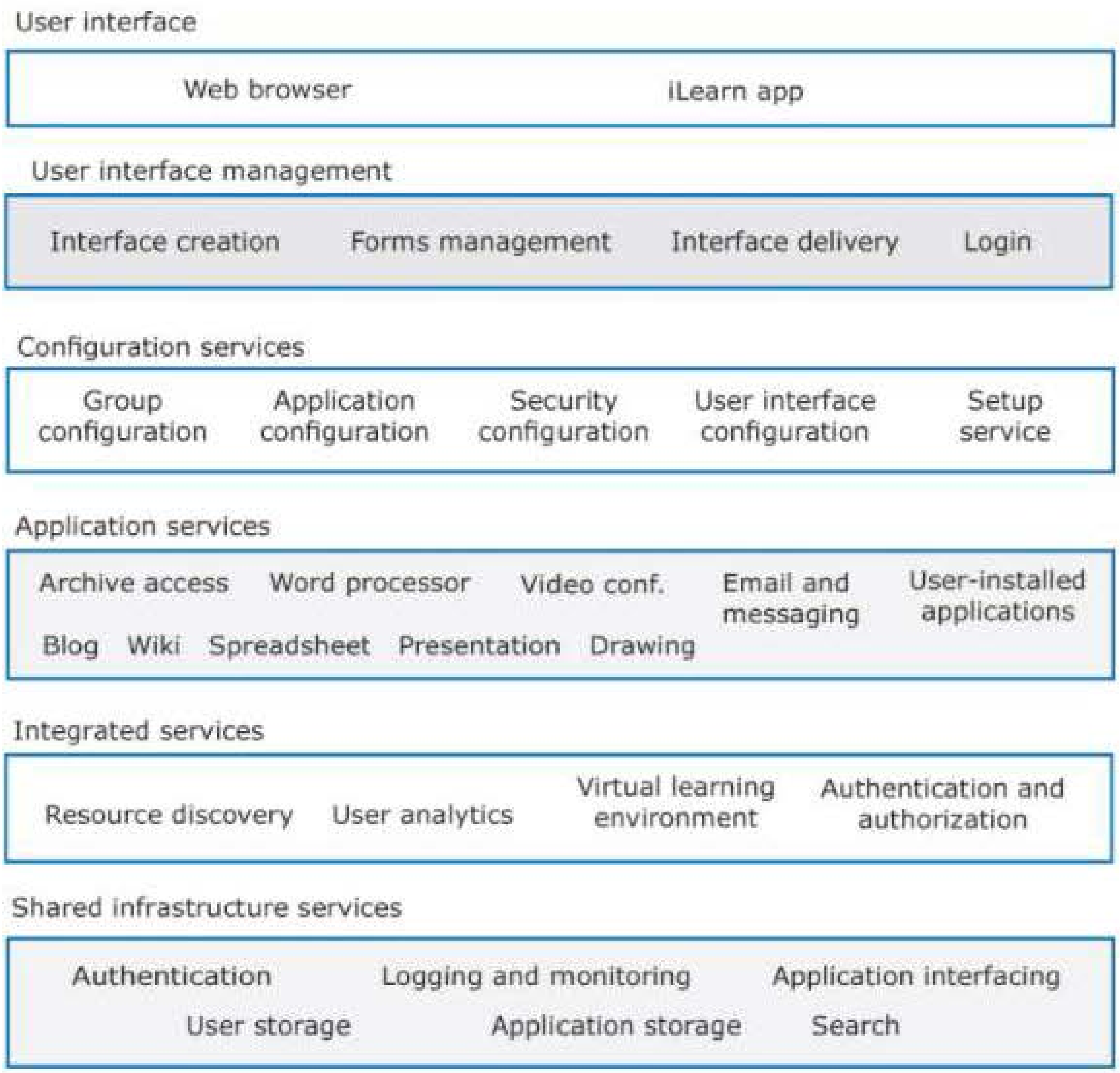


Figure 4.11
A layered architectural model of the iLearn system

To support the requirement that users should be able to configure their own version of an iLearn system, we introduced an additional layer into the system, above the application layer. This layer includes several components that incorporate knowledge of the installed applications and provide configuration functionality to end-users.

The system has a set of pre-installed application services. Additional application services can be added or existing services replaced by using the application configuration facilities. Most of these application services are independent and manage their own data. Some services are partially integrated, however, which simplifies information sharing and allows more detailed user information to be collected.

The fully integrated services have to be specially written or adapted from open-source software. They require knowledge of how the system is used and access to user data in the storage system. They may make use of other services at the same level. For example, the user analytics service provides information about how individual students use the system and can highlight problems to teachers. It needs to be able to access both log information and student records from the virtual learning environment.

System decomposition has to be done in conjunction with choosing technologies for your system (see **Section 4.5**). The reason for this is that the choice of technology used in a particular layer affects the components in the layers above. For example, you may decide to use a relational database technology as the lowest layer in your system. This makes sense if you are dealing with well-structured data. However, your decision affects the components to be included in the services layer because you need to be able to communicate with the database. You may have to include components to adapt the data passed to and from the database.

Another important technology-related decision is the interface technologies that you will use. This choice depends on whether you will be supporting browser interfaces only (often the case with business systems) or you also want to provide interfaces on mobile devices. If you are supporting mobile devices, you need to include components to interface with the relevant iOS and Android UI development toolkits.

4.4 Distribution architecture

The majority of software products are now web-based products, so they have a client–server architecture. In this architecture, the user interface is implemented on the user's own computer or mobile device. Functionality is distributed between the client and one or more server computers. During the architectural design process, you have to decide on the "distribution architecture" of the system. This defines the servers in the system and the allocation of components to these servers.

Client–server architectures are a type of distribution architecture that is suited to applications in which clients access a shared database and business logic operations on those data. **Figure 4.12** shows a logical view of a client–server architecture that is widely used in web-based and mobile software products. These applications include several servers, such as web servers and database servers. Access to the server set is usually mediated by a load balancer, which distributes requests to servers. It is designed to ensure that the computing load is evenly shared by the set of servers.

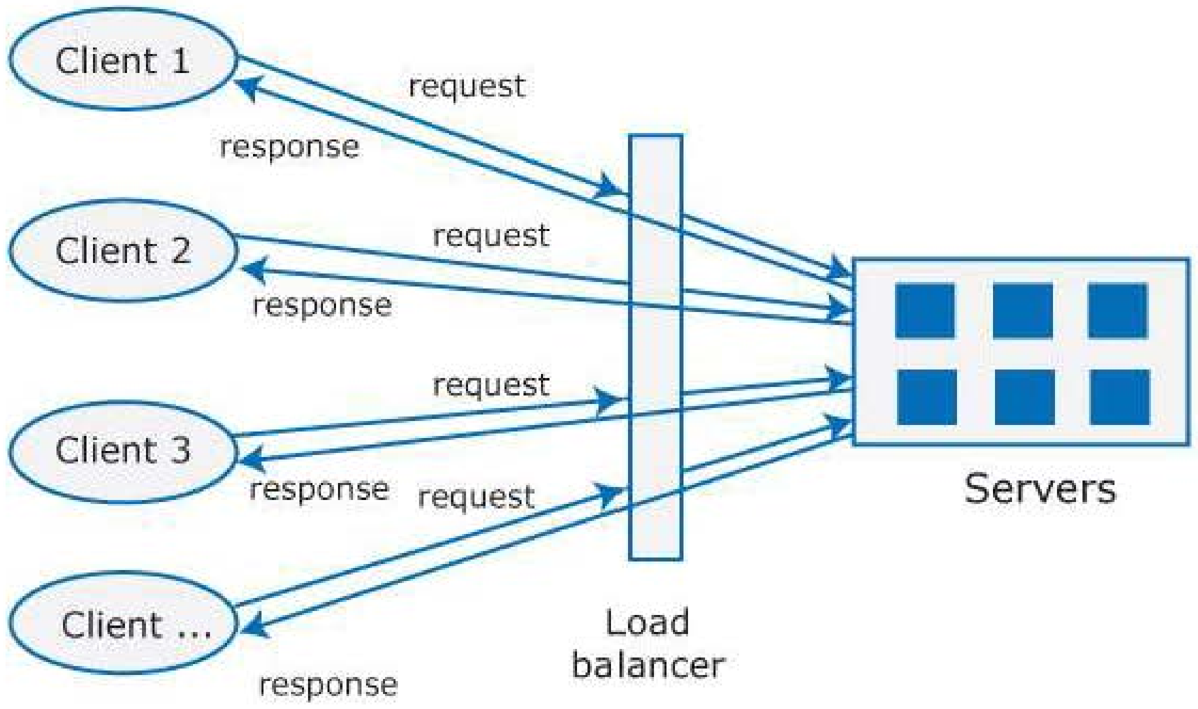


Figure 4.12
Client–server architecture

The client is responsible for user interaction, based on data passed to and from the server. When this architecture was originally devised, clients were character terminals with hardly any local processing capability. The server was a mainframe computer. All processing was carried out on the server, with the client handling only user interaction. Now clients are computers or mobile devices with lots of processing power, so most applications are designed to include significant client-side processing.

Client–server interaction is usually organized using what is called the Model-View-Controller (MVC) pattern. This architectural pattern is used so that client interfaces can be updated when data on the server change (**Figure 4.13**).

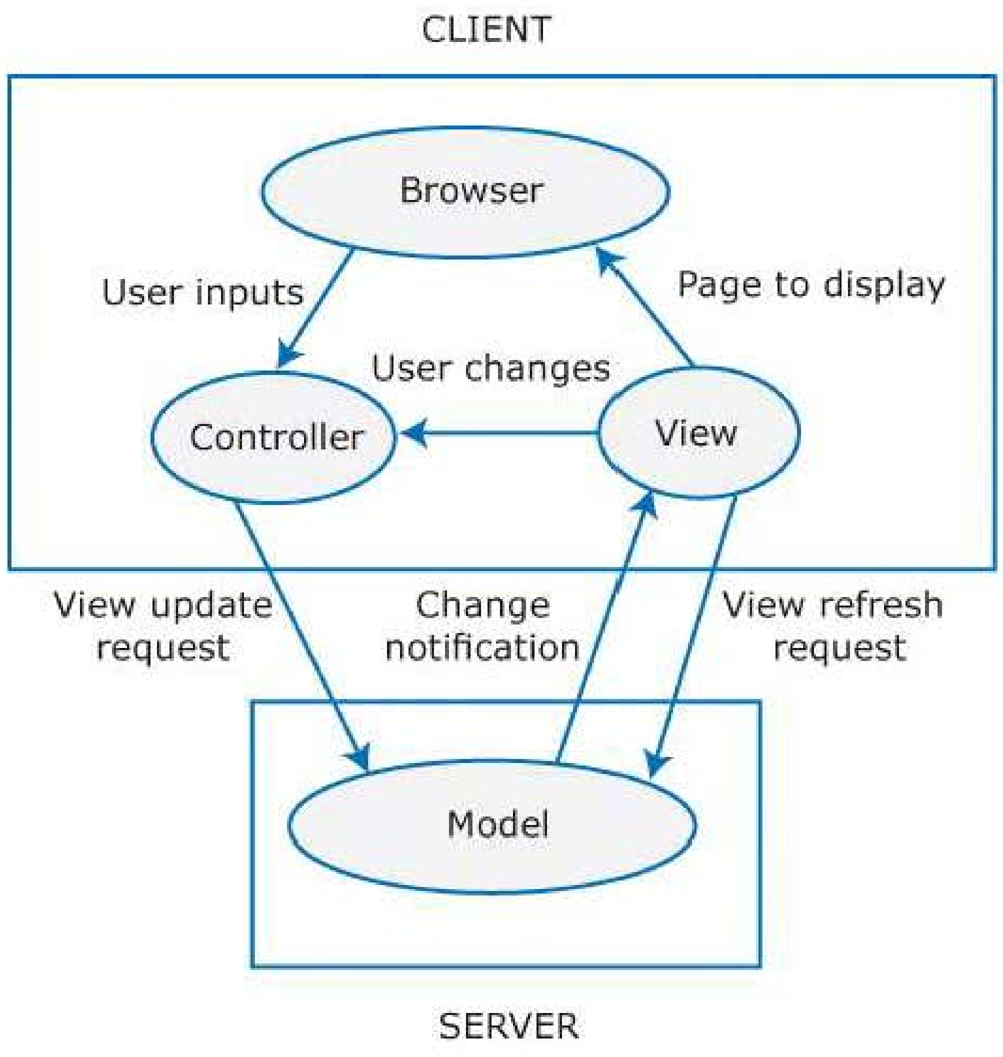


Figure 4.13
The Model-View-Controller pattern

The term "model" is used to mean the system data and the associated business logic. The model is always shared and maintained on the server. Each client has its own view of the data, with views responsible for HTML page generation and forms management. There may be several views on each client in which the data are presented in different ways. Each view registers with the model so that when the model changes, all views are updated. Changing the information in one view leads to all other views of the same information being updated.

User inputs that change the model are handled by the controller. The controller sends update requests to the model on the server. It may also be responsible for local input processing, such as data validation.

The MVC pattern has many variants. In some, all communication between the view and the model goes through the controller. In others, views can also handle user inputs. However, the essence of all of these variants is that the model is decoupled from its presentation. It can, therefore, be presented in different ways and each presentation can be independently updated when changes to the data are made.

For web-based products, Javascript is mostly used for client-side programming. Mobile apps are mostly developed in Java (Android) and Swift (iOS). I don't have experience in mobile app development, so I focus here on web-based products. However, the underlying principles of interaction are the same.

Client–server communication normally uses the HTTP protocol, which is a text-based request/response protocol. The client sends a message to the server that includes an instruction such as GET or POST along with the identifier of a resource (usually a URL) on which that instruction should operate. The message may also include additional information, such as information collected from a form. So, a database update may be encoded as a POST instruction, an identifier for the information to be updated plus the changed information input by the user. Servers do not send requests to clients, and clients always wait for a response from the server.[2]

[2]This is not strictly true if a technology such as Node.js is used to build server-side applications. This allows both clients and servers to generate requests and responses. However, the general client–server model still applies.

HTTP is a text-only protocol, so structured data must be represented as text. Two ways of representing these data are widely used—namely, XML and JSON. XML is a markup language with tags used to identify each data item. JSON is a simpler representation based on the representation of objects in the Javascript language. Usually JSON representations are more compact and faster to parse than XML text. I recommend that you use JSON for data representation.

As well as being faster to process than XML, JSON is easier for people to read. Program 4.1 shows the JSON representation of cataloging information about a software engineering textbook.

Program 4.1 An example of JSON information representation

```
{
"book": [
{
"title":  "Software Engineering",
"author": "Ian Sommerville",
"publisher": "Pearson Higher Education",
"place": "Hoboken, NJ",
 "year": "2015",
"edition": "10th",
 "ISBN": "978-0-13-394303-0"
},
 ]
}
```

There are good JSON tutorials available on the web, so I don't go into more detail about this notation.

Many web-based applications use a multi-tier architecture with several communicating servers, each with its own responsibilities. **Figure 4.14** illustrates the distributed components in a multi-tier web-based system architecture. For simplicity, I assume there is only a single instance of each of these servers and so a load balancer is not required. Web-based applications usually include three types of server:

1. A web server that communicates with clients using the HTTP protocol. It delivers web pages to the browser for rendering and processes HTTP requests from the client.
2. An application server that is responsible for application-specific operations. For example, in a booking system for a theater, the application server provides information about the shows as well as basic functionality that allows a theatergoer to book seats for shows.
3. A database server that manages the system data and transfers these data to and from the system database.

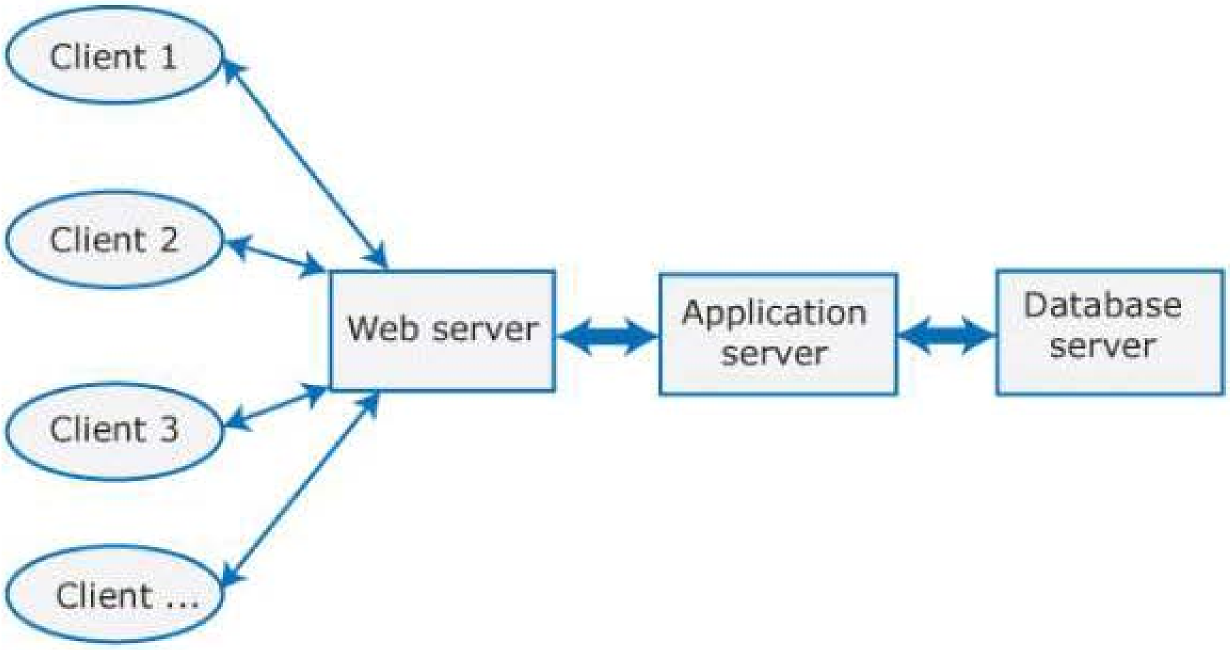


Figure 4.14
Multi-tier client–server architecture

Sometimes a multi-tier architecture may use additional specialized servers. For example, in a theater booking system, the user's payments may be handled by a credit card payment server provided by a company that specializes in credit card payments. This makes sense for most e-commerce applications, as a high cost is involved in developing a trusted payment system. Another type of specialized server that is commonly used is an authentication server. This checks users' credentials when they log in to the system.

An alternative to a multi-tier client–server architecture is a service-oriented architecture (**Figure 4.15**) where many servers may be involved in providing services. Services in a service-oriented architecture are stateless components, which means that they can be replicated and can migrate from one computer to another. A service-oriented architecture is usually easier to scale as demand increases and is resilient to failure.

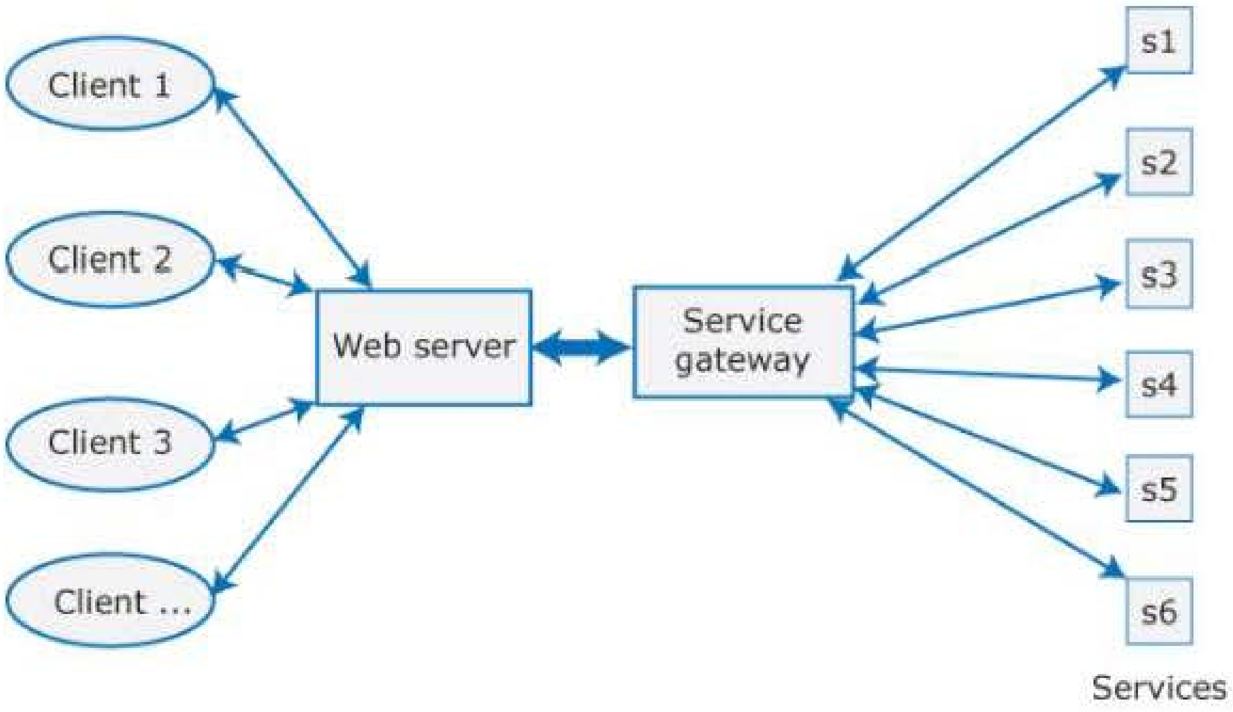


Figure 4.15
Service-oriented architecture

The services shown in **Figure 4.15** are services that support features in the system. These are the services provided by the application layer and layers above this in the decomposition stack. To keep the diagram simple, I do not show interactions between services or infrastructure services that provide functionality from lower levels in the decomposition. Service-oriented architectures are increasingly used, and I discuss them in more detail in **Chapter 6**.

We chose a service-oriented distribution architecture for the iLearn system, with each of the components shown in Figure 4.11 implemented as a separate service. We chose this architecture because we wanted to make it easy to update the system with new functionality. It also simplified the problem of adding new, unforeseen services to the system.

Multi-tier and service-oriented architectures are the main types of distribution architecture for web-based and mobile systems. You have to decide which of these to choose for your software product. The issues that you must consider are:

1. **Data type and data updates** If you are mostly using structured data that may be updated by different system features, it is usually best to have a single shared database that provides locking and transaction management. If data are distributed across services, you need a way to keep them consistent, and this adds overhead to your system.
2. **Frequency of change** If you anticipate that system components will regularly be changed or replaced, then isolating these components as separate services simplifies those changes.
3. **The system execution platform** If you plan to run your system on the cloud with users accessing it over the Internet, it is usually best to implement it as a service-oriented architecture because scaling the system is simpler. However, if your product is a business system that runs on local servers, a multi-tier architecture may be more appropriate.

When I wrote this book in 2018, the distribution architecture of most business software products was a multi-tier client–server architecture with user interaction implemented using the MVC pattern. However, these products are increasingly being updated to use service-oriented architectures, running on public cloud platforms. I think that, over time, this type of architecture will become the norm for web-based software products.

4.5 Technology issues

An important part of the process of designing software architecture is to make decisions about the technologies you will use in your product. The technologies that you choose affect and constrain the overall architecture of your system. It is difficult and expensive to change these during development, so it is important that you carefully consider your technology choices.

An advantage of product development compared to enterprise system development is that you are less likely to be affected by legacy technology issues. Legacy technologies are technologies that have been used in old systems and are still operational. For example, some old enterprise systems still rely on 1970s database technology. Modern systems may have to interact with these and this limits their design.

Unless you have to interoperate with other software products sold by your company, your choice of technology for product development is fairly flexible. **Table 4.8** shows some of the important technology choices you may have to make at an early stage of product development.

Table 4.8 Technology choices

Technology	Design decision
Database	Should you use a relational SQL data base or an unstructured NoSQL datab ase?
Platform	Should you deliver your product on a mobile app and/or a web platform?
Server	Should you use dedicated in-house s ervers or design your system to run o

	n a public cloud? If a public cloud, sh ould you use Amazon, Google, Micro soft, or some other option?
Open source	Are there suitable open-source comp onents that you could incorporate int o your products?
Development tools	Do your development tools embed ar chitectural assumptions about the so ftware being developed that limit you r architectural choices?

4.5.1 Database

Most software products rely on a database system of some kind. Two kinds of database are now commonly used: relational databases, in which the data are organized into structured tables, and NoSQL databases, in which the data have a more flexible, user-defined organization. The database has a huge influence on how your system is implemented, so which type of database to use is an important technology choice.

Relational databases, such as MySQL, are particularly suitable for situations where you need transaction management and the data structures are predictable and fairly simple. Relational databases support ACID transactions. Transactions guarantee that the database will always remain consistent even if a transaction fails, that updates will be serialized, and that recovery to a consistent state is always possible. This is really important for financial information where inconsistencies are unacceptable. So, if your product deals with financial information, or any information where consistency is critical, you should choose a relational database.

However, there are lots of situations where data are not well structured and where most database operations are concerned with reading and analyzing data rather than writing to the database. NoSQL databases, such as MongoDB, are more flexible and potentially more efficient than relational databases for this type of application. NoSQL databases allow data to be organized hierarchically rather than as flat tables, and this allows for more efficient concurrent processing of "big data."

Some applications need a mixture of both transactions and big data processing, and more of this kind of application will likely be developed in the future. Database vendors are now starting to integrate these approaches. It is likely that, during the lifetime of this book, efficient integrated database systems will become available.

4.5.2 Delivery platform

Globally, more people access the web using smartphones and tablets rather than browsers on a laptop or desktop. Most business systems are still browser-based, but as the workforce becomes more mobile, there is increasing demand for mobile access to business systems.

In addition to the obvious difference in screen size and keyboard availability, there are other important differences between developing software for a mobile device and developing software that runs on a client computer. On a phone or tablet, several factors have to be considered:

1. **Intermittent connectivity** You must be able to provide a limited service without network connectivity.
2. **Processor power** Mobile devices have less powerful processors, so you need to minimize computationally intensive operations.
3. **Power management** Mobile battery life is limited, so you should try to minimize the power used by your application.
4. **On-screen keyboard** On-screen keyboards are slow and error prone. You should minimize input using the screen keyboard to reduce user frustration.

To deal with these differences, you usually need separate browser-based and mobile versions of your product front-end. You may need a completely different decomposition architecture in these different versions to ensure that performance and other characteristics are maintained.

As a product developer, you have to decide early in the process whether you will focus on a mobile or a desktop version of your software. For consumer products, you may decide to focus on mobile delivery, but for business systems, you have to make a choice about which should be your priority. Trying to develop mobile and browser-based versions of a product at the same time is an expensive process.

4.5.3 Server

Cloud computing is now ubiquitous so a key decision you have to make is whether to design your system to run on individual servers or on the cloud. Of course, it is possible to rent a server from Amazon or some other provider, but this does not really take full advantage of the cloud. To develop for the cloud, you need to design your architecture as a service-oriented system and use the platform APIs provided by the cloud vendor to implement your software. These allow for automatic scalability and system resilience.

For consumer products that are not simply mobile apps, I think it almost always makes sense to develop for the cloud. The decision is more difficult for business products. Some businesses are concerned about cloud security and prefer to run their systems on in-house servers. They may have a predictable pattern of system usage, so there is less need to design the software to cope with large changes in demand.

If you decide to develop for the cloud, the next decision is to choose a cloud provider. The major providers are Amazon, Google, and Microsoft, but unfortunately their APIs are not compatible. This means you can't easily move a product from one to the other. The majority of consumer products probably run on Amazon's or Google's cloud, but businesses often prefer Microsoft's Azure system because of its compatibility with their existing .NET software. Alt1ernatively, there are other cloud providers, such as IBM, that specialize in business services.

4.5.4 Open source

Open-source software is software that is freely available and you can change and modify it as you wish. The obvious advantage is that you can reuse rather than implement new software, thereby reducing development costs and time to market. The disadvantages of open-source software are that you are constrained by that software and have no control over its evolution. It may be impossible to change that software to give your product a "competitive edge" over competitors that use the same software. There are also license issues that must be considered. They may limit your freedom to incorporate the open-source software into your product.

The decision about open-source software also depends on the availability, maturity, and continuing support of open-source components. Using an open-source database system such as MySQL or MongoDB is cheaper than using a proprietary database such as Oracle's database system. These are mature systems with a large number of contributing developers. You would normally only choose a proprietary database if your product is aimed at businesses that already use that kind of database. At higher levels in the architecture, depending on the type of product you are developing, fewer open-source components may be available, they may be buggy, and their continuing development may depend on a relatively small support community.

Your choice of open-source software should depend on the type of product you are developing, your target market, and the expertise of your development team. There's often a mismatch between the "ideal" open-source software and the expertise that you have available. The ideal software may be better in the long term but could delay your product launch as your team becomes familiar with it. You have to decide whether the long-term benefits justify that delay. There is no point in building a better system if your company runs out of money before that system is delivered.

4.5.5 Development technology

Development technologies, such as a mobile development toolkit or a web application framework, influence the architecture of your software. These technologies have built-in assumptions about system architectures, and you have to conform to these assumptions to use the development system. For example, many web development frameworks are designed to create applications that use the model-view-controller architectural pattern.

The development technology that you use may also have an indirect influence on the system architecture. Developers usually favor architectural choices that use familiar technologies that they understand. For example, if your team has a lot of experience with relational databases, they may argue for this instead of a NoSQL database. This can make sense, as it means the team does not have to spend time learning about a new system. It can have long-term negative consequences, however, if the familiar technology is not the right one for your software.

Key Points

- Software architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.
- The architecture of a software system has a significant influence on non-functional system properties, such as reliability, efficiency, and security.
- Architectural design involves understanding the issues that are critical for your product and creating system descriptions that show components and their relationships.
- The principal role of architectural descriptions is to provide a basis for the development team to discuss the system organization. Informal architectural diagrams are effective in architectural description because they are fast and easy to draw and share.
- System decomposition involves analyzing architectural components and representing them as a set of finer-grain components.
- To minimize complexity, you should separate concerns, avoid functional duplication, and focus on component interfaces.
- Web-based systems often have a common layered structure, including user interface layers, application-specific layers, and a database layer.
- The distribution architecture in a system defines the organization of the servers in that system and the allocation of components to these servers.
- Multi-tier client–server and service-oriented architectures are the most commonly used architectures for web-based systems.
- Making decisions about technologies such as database and cloud technologies is an important part of the architectural design process.

Recommended Reading

"Software Architecture and Design" This excellent series of articles provides sound, practical advice on general principles of software architecture and design. It includes a discussion of layered architectures in **Chapter 3**, under architectural patterns and styles. (Microsoft, 2010)

https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658093(v%3dpandp.10)

"Five Things Every Developer Should Know about Software Architecture" This is a good explanation of why designing a software architecture is consistent with agile software development. (S. Brown, 2018)

https://www.infoq.com/articles/architecture-five-things

"Software Architecture Patterns" This is a good general introduction to layered architectures, although I don't agree that layered architectures are as difficult to change as the author suggests. (M. Richards, 2015, login required)

https://www.oreilly.com/ideas/software-architecture-patterns/page/2/layered-architecture

"What is the 3-Tier Architecture?" This is a very comprehensive discussion of the benefits of using a three-tier architecture. The author argues that is isn't necessary to use more than three tiers in any system. (T. Marston, 2012)

http://www.tonymarston.net/php-mysql/3-tier-architecture.html

"Five Reasons Developers Don't Use UML and Six Reasons to Use It" This article sets out arguments for using the UML when designing software architectures. (B. Pollack, 2010)

https://saturnnetwork.wordpress.com/2010/10/22/five-reasons-developers-dont-use-uml-and-six-reasons-to-use-it/

"Mobile vs. Desktop: 10 key differences" This blog post summarizes the issues to be considered when designing mobile and desktop products. (S. Hart, 2014)

https://www.paradoxlabs.com/blog/mobile-vs-desktop-10-key-differences/

"To SQL or NoSQL? That's the Database Question" This is a good, short introduction to the pros and cons of relational and NoSQL databases. (L. Vaas, 2016)

https://arstechnica.com/information-technology/2016/03/to-sql-or-nosql-thats-the-database-question/

I recommend articles on cloud-computing and service-oriented architecture in **Chapters 5** and **6**.

Presentations, Videos, and Links

https://iansommerville.com/engineering-software-products/software-architecture

Exercises

4.1 Extend the IEEE definition of software architecture to include a definition of the activities involved in architectural design.

4.2 An architecture designed to support security may be based on either a centralized model, where all sensitive information is stored in one secure place, or a distributed model, where information is spread around and stored in many different places. Suggest one advantage and one disadvantage of each approach.

4.3 Why is it important to try to minimize complexity in a software system?

4.4 You are developing a product to sell to finance companies. Giving reasons for your answer, consider the issues that affect architectural decision making (**Figure 4.4**) and suggest which two factors are likely to be most important.

4.5 Briefly explain how structuring a software architecture as a stack of functional layers helps to minimize the overall complexity in a software product.

4.6 Imagine your manager has asked you whether or not your company should move away from informal architectural descriptions to more formal descriptions based on the UML. Write a short report giving advice to your manager. If you don't know what the UML is, then you should do a bit of reading to understand it. The article by Pollack in Recommended Reading can be a starting point for you.

4.7 Using a diagram, show how the generic architecture for a web-based application can be implemented using a multi-tier client–server architecture.

4.8 Under what circumstances would you push as much local processing as possible onto the client in a client–server architecture?

4.9 Explain why it would not be appropriate to use a multi-tier client–server architecture for the iLearn system.

4.10 Do some background reading and describe three fundamental differences between relational and NoSQL databases. Suggest three types of software product that might benefit from using NoSQL databases, explaining why the NoSQL approach is appropriate.

5 Cloud-Based Software

The convergence of powerful, multicore computer hardware and high-speed networking has led to the development of "the cloud." Put simply, the cloud is a very large number of remote servers that are offered for rent by companies that own these servers. You can rent as many servers as you need, run your software on these servers, and make them available to your customers. Your customers can access these servers from their own computers or other networked devices such as a tablet or a TV. You may rent a server and install your own software, or you may pay for access to software products that are available on the cloud.

The remote servers are "virtual servers," which means they are implemented in software rather than hardware. Many virtual servers can run simultaneously on each cloud hardware node, using virtualization support that is built in to the hardware. Running multiple servers has very little effect on server performance. The hardware is so powerful that it can easily run several virtual servers at the same time.

Cloud companies such as Amazon and Google provide cloud management software that makes it easy to acquire and release servers on demand. You can automatically upgrade the servers that you are running, and the cloud management software provides resilience in the event of a server failure. You can rent a server for a contracted amount of time or rent and pay for servers on demand. Therefore, if you need resources for only a short time, you simply pay for the time that you need.

The cloud servers that you rent can be started up and shut down as demand changes. This means that software that runs on the cloud can be scalable, elastic, and resilient (**Figure 5.1**). I think that scalability, elasticity, and resilience are the fundamental differences between cloud-based systems and those hosted on dedicated servers.

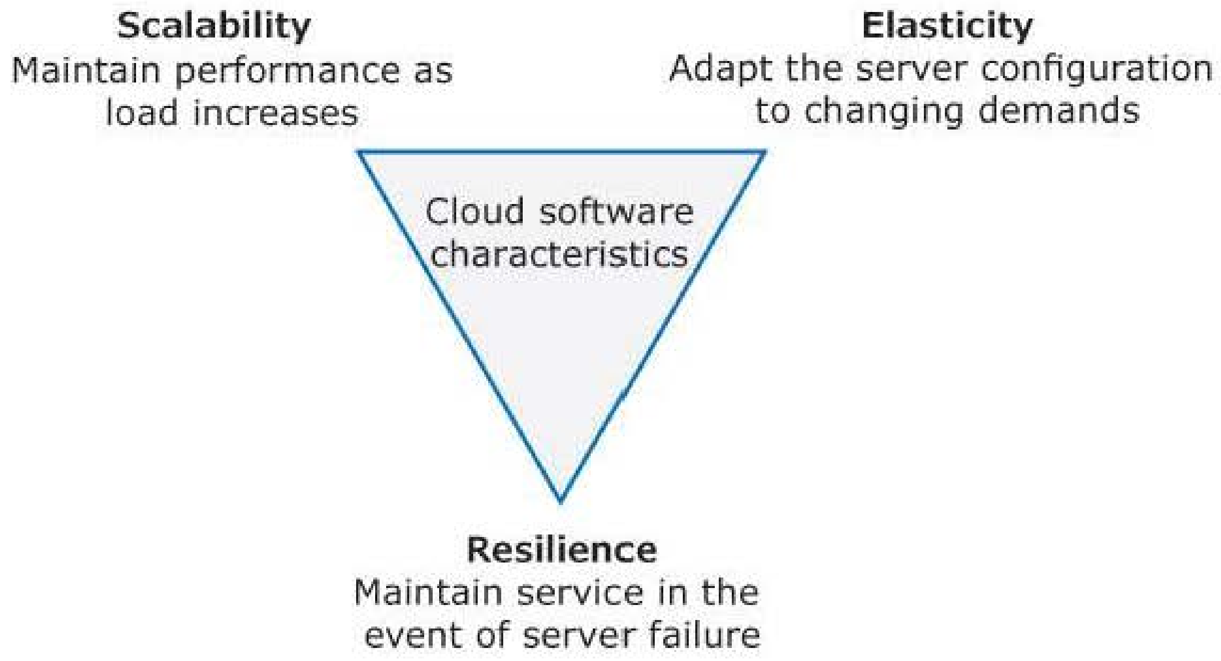

Scalability
Maintain performance as load increases
Elasticity
Adapt the server configuration to changing demands
Cloud software characteristics
Resilience
Maintain service in the event of server failure

Figure 5.1
Scalability, elasticity, and resilience

Scalability reflects the ability of your software to cope with increasing numbers of users. As the load on your software increases, the software automatically adapts to maintain the system performance and response time. Systems can be scaled by adding new servers or by migrating to a more powerful server. If a more powerful server is used, this is called scaling up. If new servers of the same type are added, this is called scaling out. If your software is scaled out, copies of your software are created and executed on the additional servers.

Elasticity is related to scalability but allows for scaling down as well as scaling up. That is, you can monitor the demand on your application and add or remove servers dynamically as the number of users changes. This means that you pay for only the servers you need, when you need them.

Resilience means that you can design your software architecture to tolerate server failures. You can make several copies of your software available concurrently. If one of these fails, the others continue to provide a service. You can cope with the failure of a cloud data center by locating redundant servers in different places.

If you are setting up a new software product company or development project, it isn't cost effective to buy server hardware to support software development. Rather, you should use cloud-based servers that are accessed from your development machines. The benefits of adopting this approach rather than buying your own servers are shown in **Table 5.1**.

Table 5.1 Benefits of using the cloud for software development

Factor	Benefit
Cost	You avoid the initial capital costs of hardware procurement.
Startup time	You don't have to wait for hardware to be delivered before you can start work. Using the cloud, you can have servers up and running in a few minutes.
Server choice	If you find that the servers you are renting are not powerful enough, you can upgrade to more powerful systems. You can add servers for short-term requirements, such as load testing.
Distributed development	If you have a distributed development team, working from different locations, all team members have the same development environment and can seamlessly share all information.

I recommend that using the cloud for both development and product delivery, as a cloud service, should be the default choice for new software product development. The only exceptions might be if you are delivering products for a specialized hardware platform, or your customers have security requirements that forbid the use of external systems.

All major software vendors now offer their software as cloud services. Customers access the remote service through a browser or mobile app rather than installing it on their own computers. Well-known examples of software delivered as a service include mail systems such as Gmail and productivity products such as Office 365.

This chapter introduces some fundamental ideas about cloud-based software that you have to consider when making architectural decisions. I explain the idea of containers as a lightweight mechanism to deploy software. I explain how software can be delivered as a service, and I introduce general issues in cloud-based software architecture design. **Chapter 6** focuses on a service-oriented architectural pattern that is particularly relevant to cloud-based systems—namely, microservices architecture.

5.1 Virtualization and containers

All cloud servers are virtual servers. A virtual server runs on an underlying physical computer and is made up of an operating system plus a set of software packages that provide the server functionality required. The general idea is that a virtual server is a stand-alone system that can run on any hardware in the cloud.

This "run anywhere" characteristic is possible because the virtual server has no external dependencies. An external dependency means you need some software, such as a database management system, that you are not developing yourself. For example, if you are developing in Python, you need a Python compiler, a Python interpreter, various Python libraries, and so on.

When you run software on different computers, you often encounter problems because some of the external software that you rely on is unavailable or is different in some way from the version that you're using. If you use a virtual server, you avoid these problems. You load all of the software that you need, so you are not relying on software being made available by someone else.

Virtual machines (VMs), running on physical server hardware, can be used to implement virtual servers (**Figure 5.2**). The details are complex, but you can think of the hypervisor as providing a hardware emulation that simulates the operation of the underlying hardware. Several of these hardware emulators share the physical hardware and run in parallel. You can run an operating system and then install server software on each hardware emulator.

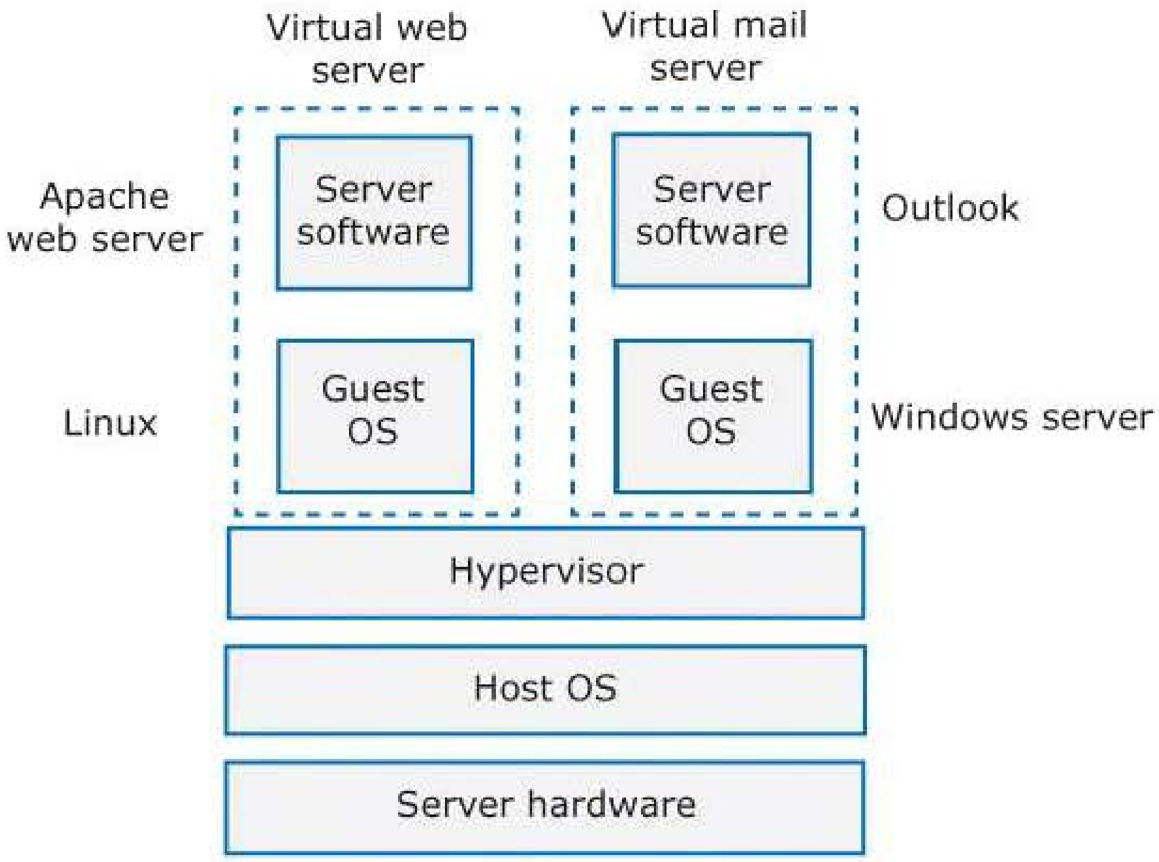


Figure 5.2
Implementing a virtual server as a virtual machine

The advantage of using a virtual machine to implement virtual servers is that you have exactly the same hardware platform as a physical server. You can therefore run different operating systems on virtual machines that are hosted on the same computer. For example, **Figure 5.2** shows that Linux and Windows can run concurrently on separate VMs. You may want to do this so that you can run software that is available for only one particular operating system.

The problem with implementing virtual servers on top of VMs is that creating a VM involves loading and starting up a large and complex operating system (OS). The time needed to install the OS and set up the other software on the VM is typically between 2 and 5 minutes on public cloud providers such as AWS. This means that you cannot instantly react to changing demands by starting up and shutting down VMs.

In many cases, you don't really need the generality of a virtual machine. If you are running a cloud-based system with many instances of applications or services, these all use the same operating system. To cater to this situation, a simpler, lightweight, virtualization technology called "containers" may be used.

Using containers dramatically speeds up the process of deploying virtual servers on the cloud. Containers are usually megabytes in size, whereas VMs are gigabytes. Containers can be started up and shut down in a few seconds rather than the few minutes required for a VM. Many companies that provide cloud-based software have now switched from VMs to containers because containers are faster to load and less demanding of machine resources.

Containers are an operating system virtualization technology that allows independent servers to share a single operating system. They are particularly useful for providing isolated application services where each user sees their own version of an application. I show this in **Figure 5.3**, where a graphics design system product uses basic graphics libraries and a photo management system. A container with the graphics support software and the application is created for each user of the software.